Tracking and Mitigation of Malicious Remote Control Networks

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Thorsten Holz

aus Trier

Mannheim, 2009

Dekan:	Prof. Dr. Felix Christoph Freiling, Universität Mannheim
Referent:	Prof. Dr. Felix Christoph Freiling, Universität Mannheim
Korreferent:	Prof. Dr. Christopher Krügel, University of California, Santa Barbara

Tag der mündlichen Prüfung: 30. April 2009

Abstract

Attacks against end-users are one of the negative side effects of today's networks. The goal of the attacker is to compromise the victim's machine and obtain control over it. This machine is then used to carry out denial-of-service attacks, to send out spam mails, or for other nefarious purposes. From an attacker's point of view, this kind of attack is even more efficient if she manages to compromise a large number of machines in parallel. In order to control all these machines, she establishes a *malicious remote control network*, i.e., a mechanism that enables an attacker the control over a large number of compromised machines for illicit activities. The most common type of these networks observed so far are so called *botnets*.

Since these networks are one of the main factors behind current abuses on the Internet, we need to find novel approaches to stop them in an automated and efficient way. In this thesis we focus on this open problem and propose a general root cause methodology to stop malicious remote control networks. The basic idea of our method consists of three steps. In the first step, we use *honeypots* to collect information. A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource. This technique enables us to study current attacks on the Internet and we can for example capture samples of autonomous spreading malware (*malicious software*) in an automated way. We analyze the collected data to extract information about the remote control mechanism in an automated fashion. For example, we utilize an automated binary analysis tool to find the Command & Control (C&C) server that is used to send commands to the infected machines. In the second step, we use the extracted information to infiltrate the malicious remote control networks. This can for example be implemented by impersonating as a bot and infiltrating the remote control channel. Finally, in the third step we use the information collected during the infiltration phase to mitigate the network, e.g., by shutting down the remote control channel such that the attacker cannot send commands to the compromised machines.

In this thesis we show the practical feasibility of this method. We examine different kinds of malicious remote control networks and discuss how we can track all of them in an automated way. As a first example, we study botnets that use a central C&C server: We illustrate how the three steps can be implemented in practice and present empirical measurement results obtained on the Internet. Second, we investigate botnets that use a peer-to-peer based communication channel. Mitigating these botnets is harder since no central C&C server exists which could be taken offline. Nevertheless, our methodology can also be applied to this kind of networks and we present empirical measurement results substantiating our method. Third, we study *fast-flux service networks*. The idea behind these networks is that the attacker does not directly abuse the compromised machines, but uses them to establish a proxy network on top of these machines to enable a robust hosting infrastructure. Our method can be applied to this novel kind of malicious remote control networks and we present empirical results supporting this claim. We anticipate that the methodology proposed in this thesis can also be used to track and mitigate other kinds of malicious remote control networks.

Zusammenfassung

Angriffe gegen Nutzer sind einer der negativen Seiteneffekte heutiger Netze. Das Ziel eines Angreifers ist es, die Maschine des Opfers zu kompromittieren und Kontrolle über diese zu erlangen. Die Maschine wird dann dazu benutzt, *Denial-of-Service* Angriffe durchzuführen, Spam-Nachrichten zu verschicken oder für weitere schädliche Zwecke. Aus der Sicht eines Angreifers sind solche Angriffe noch effizienter, wenn er es schafft, eine große Anzahl an Maschinen gleichzeitig zu kompromittieren. Um alle diese Maschinen kontrollieren zu können, setzt der Angreifer ein *malicious remote control network* auf, das heisst einen Mechanismus, der es dem Angreifer erlaubt, eine große Anzahl an kompromittierten Maschinen zu kontrollieren, um diese für rechtswidrige Aktionen zu benutzen. Die bekannteste Art dieser Netze sind sogenannte *Botnetze*.

Weil diese Netze einer der Hauptfaktoren derzeitiger Missbräuche im Internet sind, benötigen wir neuartige Ansätze, um sie in einem automatisierten und effizienten Prozess stoppen zu können. In dieser Arbeit konzentrieren wir uns auf dieses offene Problem und stellen eine allgemeine Methodik vor, um die Grundursache hinter *malicious remote control networks* zu stoppen. Die Grundidee unserer Methodik besteht aus drei Schritten. Im ersten Schritt benutzen wir *Honeypots*. Ein Honeypot ist ein Informationssystem, dessen Funktion darin besteht, von Angreifern auf unerlaubte oder nicht autorisierte Weise benutzt zu werden. Diese Technik erlaubt es uns, gegenwärtige Angriffe im Internet zu erforschen und wir können beispielsweise Kopien von Schadsoftware (*malware*), die sich selbständig verbreitet, in einem automatisierten Prozess sammeln. Wir analysieren die gesammelten Daten in einem automatisierten Verfahren, um Informationen über den Fernsteuerungsmechanismus zu extrahieren. Wir benutzen beispielsweise ein automatisiertes Programm zur Analyse von Binärdateien, um den *Command & Control* (C&C) Server zu finden, mit dessen Hilfe Kommandos zu den infizierten Maschinen gesendet werden. Im zweiten Schritt benutzen wir die extrahierten Informationen, um das *malicious remote control network* zu infiltrieren. Dies kann beispielsweise umgesetzt werden, indem wir das Verhalten eines Bots simulieren und uns in den Fernsteuerungskanal einschleusen. Letztendlich benutzen wir im dritten Schritt die in der Infiltrierungsphase gesammelten Informationen, um das Netz abzuschwächen. Dies geschieht beispielsweise, indem der Kommunikationskanal geschlossen wird, so dass der Angreifer keine Befehle mehr zu den kompromittierten Maschinen senden kann.

In dieser Arbeit demonstrieren wir die praktische Umsetzbarkeit der vorgeschlagenen Methodik. Wir untersuchen verschiedene Typen von *malicious remote control networks* und erörtern, wie wir alle auf automatisierte Art und Weise aufspüren können. Als erstes Beispiel untersuchen wir Botnetze mit einem zentralen C&C-Server: Wir erläutern, wie die vorgeschlagenen drei Schritte in der Praxis umgesetzt werden können und präsentieren empirische Messergebnisse, die im Internet gesammelt wurden. Zweitens erforschen wir Botnetze mit einem Peer-to-Peer-basierten Kommunikationskanal. Eine Abschwächung dieser Botnetze ist schwieriger, da kein zentraler C&C-Server existiert, der abgeschaltet werden könnte. Dennoch kann unsere Methodik auch auf diese Art von Netzen angewandt werden und wir stellen empirische Messergebnisse vor, die unsere Methode untermauern. Als dritten Fall analysieren wir *fast-flux service networks*. Die

Idee hinter diesen Netzen ist, dass der Angreifer die kompromittierten Maschinen nicht direkt missbraucht. Stattdessen benutzt er sie dazu, ein Proxy-Netzwerk mit Hilfe dieser Maschinen aufzubauen, das dann eine robuste Hostinginfrastruktur ermöglicht. Unsere Methodik kann auch auf diese neuartige Art von *malicious remote control networks* angewandt werden und wir präsentieren empirische Ergebnisse, die diese Behauptung bestätigen. Wir erwarten, daß die in dieser Arbeit vorgeschlagene Methodik auch dazu benutzt werden kann, weitere Arten von *malicious remote control networks* zu verfolgen und abzuschwächen.

Acknowledgements

A thesis is the result of a longer process and often involves collaboration with many different people. I would like to take the opportunity and acknowledge in this section all the individuals who helped me with the work on my thesis during the past years.

First of all, I would like to express my gratitude to my advisor, Prof. Dr. Felix Freiling, for making this thesis possible. He opened me a door to the fascinating world of computer science and shared his insight and wisdom with me. His supervision, advice, and guidance from the early stages of this research on helped me to complete this work. In particular, the discussion about methodologically sound approaches were always insightful and taught me scientific thinking.

I would also like to thank Prof. Dr. Christopher Krügel for his inspiring work and the interesting discussions we had. Our research interests overlap to a large degree and his many impressive publications stimulated my research interests over the years. I am really looking forward to work closely together with him in the near future. There are still lots of open problems and opportunities for further work in this area.

In addition, I would like to thank all members of the Laboratory for Dependable Distributed Systems. Without the support from Maximillian Dornseif, Martin Mink, Lucia Draque Penso Rautenbach, Zina Benenson, Michael Becher, Carsten Willems, Christian Gorecki, Philipp Trinius, Markus Engelberth, and Jan Göbel, there would not have been such a pleasant and prolific working atmosphere at the Lab. Especially working together with Max and Carsten was always inspiring and led to many interesting projects and insightful discussions. And traveling together with them was also lots of fun! Presumably I traveled a bit more than the average PhD student, but I really enjoyed discovering many parts of the world and getting to know different cultures.

In addition, acknowledgments are also owed to Sabine Braak and Jürgen Jaap for their indispensable help dealing with travel funds, computer hardware, administration, and all bureaucratic matters during my work. Furthermore, I would also like to thank all diploma students and student workers, who helped over the years to implement some parts of the work presented in this thesis. In particular, Ben Stock and Matthias Luft helped a lot to build and maintain the analysis environment.

I also benefited from collaboration with many people from all over the world, which lead to parts of the work presented in this thesis. First and foremost, I would like to thank Niels Provos for the collaboration on the *Virtual Honeypots* book: although it was sometimes exhausting, writing a book was a valuable experience. I also enjoyed the collaboration with Moritz Steiner, Konrad Rieck, Engin Kirda, Jose Nazario, Rainer Böhme, Pavel Laskov, David Dagon, Peter Wurzinger, Leyla Bilge, Jianwei Zhuge, and many others. This cooperation lead to several publications, which were used as basic material for this thesis. In addition, I had several interesting discussions with Robin Sommer and Christian Kreibich — I am sure that a paper will follow eventually. Besides the academic point of view, I also enjoyed the collaboration with many people from the security industry: leaving the ivory tower and attending "hacker conferences" was always inspiring and lead to many novel insights. In particular, the collaboration with William and Robert was always fascinating and I was quite often deeply impressed by

their work. Furthermore, I would like to thank Dragos Ruiu and the whole SecWest staff for organizing all the conferences. Many other people provided information useful for my research, especially different mailing lists were invaluable for my progress in different areas. Another important factor for my thesis was the collaboration with Sunbelt Software, especially Chad Loeven and Erik Sites, who provided me with interesting information and the opportunity to bring some results of this thesis to market. Presumably I forgot several people whom I should thank — I hope you do not mind.

And there are many other people who helped me during the time of this thesis. First and foremost, special thanks to all members of the German and French Honeynet Project. In particular, the *Nepenthes* development team, Georg Wicherski, Laurent Oudot, Frederic Raynal, Fabien Pouget, and several other people from both projects, helped me a lot and offered me the possibility to discuss ideas and share information. This work also benefited from the discussions with members of the Honeynet Project, especially Lance Spitzner, David Watson, Julian Grizzard, Chris Lee, David Dittrich, Ron Dodge, and several others. At the begin of my PhD studies I stayed for one month at Georgia Tech, which was only possible due to the assistance of Julian and Chris.

Special thanks go to all members of the Center for Computing and Communication at RWTH Aachen University, in particular to Jens Hektor, Thomas Bötcher, Andreas Schreiber, and Silvia Leyer. Without their help and the infrastructure they provided, many results of this thesis would not have been possible.

Last, but certainly not least, I would like to thank my parents Ewald and Mathilde Holz, my brother Volker, my uncle Alfons, and my aunt Margret for their life-long love and support. I especially owe much to my parents for their assistance through all these years — they are the ones who made my education possible in the first place. Without Andrea and her love and tireless support (especially her culinary skills), this work would not have been possible. She shared with me all the emotional dynamics during the last few years and brightened up my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The increasing professionalism in cybercrime. Today, we continue to observe a major trend in the area of Internet security: attacks are becoming increasingly dangerous and devastating. This is not only due to the exponential growth of the Internet population and the increasing financial value which Internet transactions have nowadays. It is also a sign that attackers plan and coordinate their deeds with more criminal energy and conviction: it seems like a whole underground economy is prospering, in which cybercrime plays a major role [MT06, FPPS07, HEF08]. As an example, we consider the problem of unsolicited bulk email, commonly denoted as *spam*. This phenomenon is a relatively old and well-known problem in this direction, but it has turned much more dangerous lately through *phishing*. The term phishing (composition of *password harvesting fishing*) describes scam emails that trick recipients into revealing sensitive information by masquerading as a trustworthy brand. Furthermore, spam messages that contain malicious attachments are another popular attack vector nowadays.

Another important witness of the increasing professionalism in Internet crime are so called *Denial-of-Service* (DoS) attacks. A DoS attack is an attack on a computer system or network that causes a loss of service to users, typically the loss of network connectivity and services by consuming the bandwidth of the victim network or overloading the computational resources of the victim system [MDDR04]. Using available tools [Dit09], it is relatively easy to mount DoS attacks against remote networks. For the (connection-oriented) Internet protocol TCP, the most common technique is called *TCP SYN flooding* [Com96, SKK+97] and consists of creating a large number of "half open" TCP connections on the target machine, thereby exhausting kernel data structures and making it impossible for the machine to accept new connections. For the (connectionless) protocol UDP, the technique of *UDP flooding* consists of overrunning the target machine with a large number of UDP packets, thus consuming the network bandwidth and other computational resources of the victim.

Distributed Denial-of-Service attacks as a major threat. Like spam, it is well-known that DoS attacks are extremely hard to prevent because of their "semantic" nature. In the terminology of Schneier [Sch00], semantic attacks target the way we assign meaning to content. For example, it is very hard to distinguish a DoS attack from a peak in the popularity of a large website. Using authentication it is in principle possible to detect and identify the single origin of a DoS attack by looking at the distribution of packets over IP addresses. However, it is almost impossible to detect such an attack if multiple attack hosts act in a coordinated fashion against their victim. Such attacks are called *Distributed Denial-of-Service* (DDoS). DDoS attacks are one of the most dangerous threats on the Internet today since they are not limited to web servers: virtually any service available on the Internet can be the target of such an attack. Higher-level protocols can be used to increase the load even more effectively by using very specific attacks, such as running exhausting search queries on bulletin boards or mounting *web spidering attacks*, i.e., starting from a given website and then recursively requesting all links on that site.

In the past, there are several examples of severe DDoS attacks, for which we only want to present a few representative examples. In February 2000, an attacker targeted major e-commerce companies and news-sites [Gar00]. The network traffic flooded the available Internet connection so that no users could access these websites for several hours. It is estimated that online shops like Amazon loose about $550,000 for every hour that their website is not online [Pat02], and thus the financial damage due to this kind of attacks can be high. In the recent years, the threat posed by DDoS attacks grew and began to turn into real cybercrime. An example of this professionalism are blackmail attempts against a betting company during the European soccer championship in 2004 [New04]. The attacker threatened to take the website of this company offline unless the company payed money. Similar documented cybercrime cases happened during other major sport events and against gambling websites, in which the attackers are said to have extorted millions of dollars [Sop06]. Furthermore, paid DDoS attacks to take competitor's websites down were reported in 2004 [Fed04]. Also DDoS attacks with a suspected political background were observed: in April/May 2007, attacks against Estonian government sites took place [Naz07] and in July/August 2008 several servers of the Georgian Republic were attacked [Naz08]. Several DDoS attacks were observed that target the core infrastructure of the Internet, namely the Root servers of the Domain Name System (DNS): Figure 1.1 shows an example of an attack against the G-root server which took place in February 2007, in which the reachability of this server was severely affected for several hours. This shows that the attackers can use (and actually perform) DDoS attacks to affect the Internet's core infrastructure. This kind of attacks could – if lasting for a longer time – affect virtually any systems connected to the Internet since DNS is an integral part of today's networks.

To summarize, denial-of-service attacks pose a severe threat to today's networks, especially when the attack is distributed across many different machines. It is thus important to address the root cause of these attacks to find novel ways to stop them.

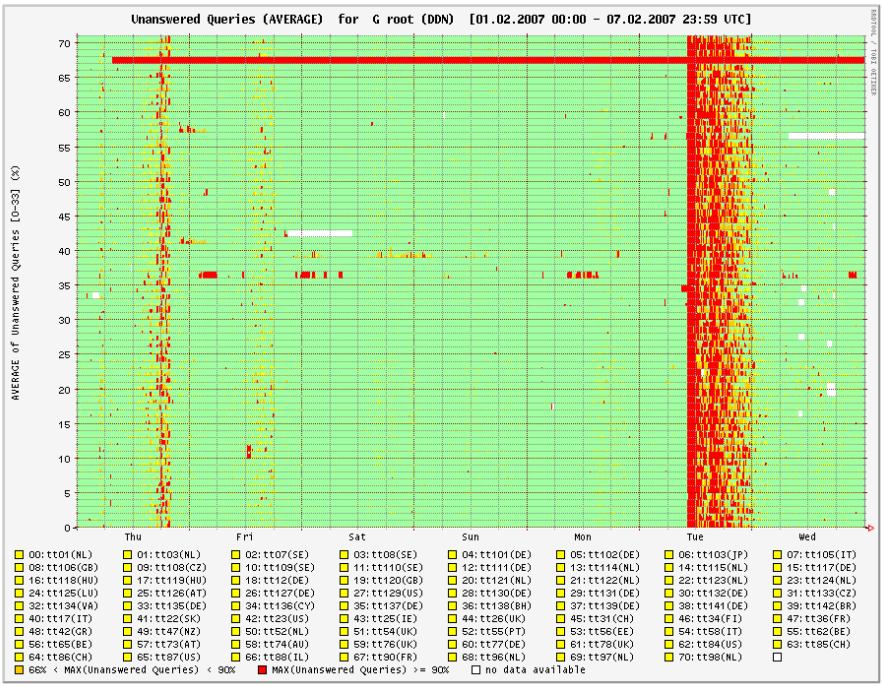


Figure 1.1: Distributed Denial-of-Service attack against G-root DNS server in February 2007, severely affecting the reachability of this server [RIP07].

1.2 Topic of Thesis

Malicious remote control networks as a tool for DDoS and other attacks. The main mechanism to launch DDoS attacks today are so called *botnets* [McC03a, Hon05], i.e., networks of compromised machines that are remotely controlled by an attacker. Botnets often consist of several thousand infected machines and enable an attacker to cause serious damage. Botnets are regularly used for DDoS attacks since their combined bandwidth overwhelms the available bandwidth of most target systems. In addition, several thousand compromised machines can generate so many packets per second that the target is unable to respond to so many requests and also the network infrastructure can be affected by such attacks. Besides DDoS attacks, botnets are also commonly used by the attackers (so called *botherder* or *botmaster*) to send spam or phishing mails, mass identity theft, and similar illicit activities.

In more general terms, attackers use compromised machines to build up *malicious remote control networks*. Throughout the rest of this thesis, we use the following definition for this kind of networks:

> A malicious remote control network is a mechanism that enables an attacker the control over a large number of compromised machines for illicit activities.

Examples of malicious remote control networks include "classical" botnets that are controlled using different communication mechanisms like IRC or peer-to-peer protocols. Another example are *fast-flux service networks*, in which the attacker does not execute commands on the infected machines, but establishes a proxy network on top of the compromised machines. A third example are *spam botnets*, i.e., networks of compromised machines which are solely used for sending spam mails.

This thesis investigates the phenomenon of malicious remote control networks and discusses the detection, analysis, and mitigation of these networks. It thus aims at understanding and countering a major threat to today's critical Internet infrastructure.

1.3 Contributions

This thesis makes the following contributions:

1.3.1 General Method to Prevent Malicious Remote Control Networks

We need efficient methods to stop malicious remote control networks due to the severity of the problems caused by these networks (e.g., DDoS attacks, spam mails, and mass identity theft). In this thesis, we introduce a novel and general root cause methodology to prevent this threat. We show how different techniques from the area of *honeypots* can be used to achieve this goal. A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource [Mai03]. The basic idea of the proposed methodology is that we can use honeypots to collect information about current attack vectors in a fully automated way and thus learn more about malicious remote control networks. Based on the collected information, we can then infiltrate the network in an efficient and automated way. Finally, we can mitigate the whole network based on the insights and the information collected during the infiltration phase, and thus prevent the threat. Throughout this thesis, we provide evidence of the feasibility of this approach based on real-world measurements on the Internet that result in a couple of additional contributions.

Figure 1.2 provides an overview of the different kinds of malicious remote control networks we study in this thesis. The *communication direction* describes the mechanism that is used by an infected machine to receive commands from the attacker. On the one hand, this mechanism can be *push-based* if the attacker sends the command, e.g., to all bots within an IRC channel. On the other hand, the infected machines can periodically *pull* for new commands and send requests to the attacker. Orthogonal to the communication direction is the *communication architecture*, which describes the actual communication structure. This can be a *centralized* model in which the attacker uses one central server to which infected machines connect. In contrast, the architecture can also be implemented as a *peer-to-peer* system. Figure 1.2 also provides an example for each of the four combinations of communication mechanisms. No example for a malicious remote control network that uses a push-based, peer-to-peer communication channel

Com. architecture / Communication direction	Central communication	Peer-to-peer communication
Push mechanism	IRC botnets (Chap. 4)	Flooding
Pull mechanism	HTTP botnets (Chap. 4) Fast-flux service networks (Chap. 6)	Publish/subscribe-style communication (Chap. 5)

Figure 1.2: Overview of different communication mechanisms used in malicious remote control networks.

exists in the wild and thus we focus in this thesis on the remaining three communication classes. In the following, we explain the contributions of each aspect in more detail.

1.3.2 Tracking Botnets with Central C&C Server

It may seem unlikely that it is possible to automatically analyze and infiltrate a malicious remote control method crafted by attackers for evil purposes. However, we provide evidence of the feasibility of our strategy by describing how we successfully tracked and investigated the automated attack activity of botnets on the Internet. We show how botnets that use a central *Command and Control* (C&C) server can be prevented using our methodology. Such botnets can use two different communication mechanisms to distribute the attacker's commands: on the one hand, botnets can implement a *push* mechanism in which the attacker pushes the command to each infected machine. This is the common setup for IRC-based botnets and we cover this type of botnets in detail in Chapter 4. On the other hand, botnets can implement a *pull* mechanism like for example HTTP-based botnets do: periodically, each bot requests a specific URL, in which it encodes status information, from the C&C server. As a reply, the server sends to the infected machine the command it should execute. In this thesis, we show how our methodology can be used to prevent both kinds of botnets.

1.3.3 Tracking Botnets with Peer-to-Peer-based C&C Server

Botnets with a central server have a single point of failure: once the central server is offline, the whole botnet is non-functional since the attacker cannot send commands to the infected machines. From an attacker's point of view, it is thus desirable to have a more robust communication mechanism within the botnet. As a result, new botnet structures emerged that use peer-to-peer-based communication protocols: each infected machine is a peer that can relay messages and act as a client and server. Such botnets are harder to mitigate since no central C&C server can be taken offline.

In this thesis, we introduce a novel mechanism to mitigate botnets that use publish/subscribe-style communication: in such systems the network nodes do not directly *send* information to each other. Instead, an information provider *publishes* a piece of information i, e.g., a file, using an identifier which is derived solely from i. An information consumer can then *subscribe* to certain information using a filter on such identifiers. Such a communication mechanism is commonly used in the peer-to-peer botnets observed up to now. We show that the methodology proposed in this thesis can be used to prevent botnets that use publish/subscribe-style communication and present empirical measurement results for Storm Worm, the most prominent peer-to-peer botnet in the wild observed up to now.

1.3.4 Tracking Fast-Flux Service Networks

Besides using the infected machines for sending spam mails or performing DDoS attacks, an attacker can also use malicious remote control networks for other purposes. An emerging threat are so called *fast-flux service networks* (FFSNs). The basic idea of such networks is that the attacker establishes a distributed proxy network on top of compromised machines that redirects traffic through these proxies to a central site, which hosts the actual content the attacker wants to publish. The focus of FFSNs thus lies in constructing a robust hosting infrastructure with the help of the victim's machines. In contrast to "classical" botnets, FFSNs are thus a different kind of malicious remote control network.

We show in this thesis that the proposed methodology can also be used to combat this novel threat. We provide a detailed overview of fast-flux service networks and study their essential properties and design principles. Based on these features, we develop a metric to identify FFSNs in an efficient way. Furthermore, we discuss how these networks can be prevented based on our methodology and present empirical measurement results collected on the Internet.

1.4 Thesis Outline

This thesis is subdivided into six chapters. In Chapter 2, we provide an overview of different concepts in the area of honeypots and botnets since these two approaches are the basic concepts used during this thesis. Furthermore, we review related work and discuss how the work presented in this thesis contributes to the state-of-the-art in the areas of honeypot- and botnet-related research.

Chapter 3 introduces a general root cause methodology to prevent malicious remote control networks. The basic idea of the approach is to first infiltrate the network, then analyze the network from the inside, and finally use the collected information to stop it. This discussion serves as the foundation of this thesis and we show in the following three chapters empirical realizations of the methodology.

In Chapter 4, we first show how the methodology can be applied to botnets with a central C&C server. The infiltration step itself is split into three phases. First, we

show how we can use honeypots to automatically and efficiently collect samples of malicious software (*malware*). Second, we introduce an approach to automatically analyze the collected binaries. Third, we show how the actual infiltration process can be automated to a high degree by mimicking the behavior of a real bot. We exemplify how this approach can be used for botnets that use either IRC or HTTP as communication protocol. Empirical measurement results from a university environment confirm the efficiency of the approach presented in this thesis.

Chapter 5 extends the botnet tracking methodology from botnets with a central server to botnets with a peer-to-peer-based communication channel. We show how this kind of botnets can be prevented with our approach, confirming that the proposed methodology is general. As a case study, we focus on Storm Worm, the most prominent peer-to-peer-based botnet observed in the wild today. We present a detailed overview of the botnet and show empirical measurement results which confirm that our approach can be successfully applied in practice.

Chapter 6 focusses on a different kind of malicious remote control networks, namely fast-flux service networks. This kind of networks enables an attacker to use a large number of infected machines to act as proxies, using the compromised machines to host illicit content. We first provide a detailed overview of fast-flux service networks and then study how this threat can be prevented. Again, we use empirical measurements to substantiate the practicability of our approach.

Finally, Chapter 7 summarizes the thesis and concludes with an overview of future research directions in this area.

1.5 List of Publications

This thesis is a monograph which contains some unpublished material, but is mainly based on the publications listed in this section.

Chapter 2 is based on a book published together with Provos on virtual honeypots [PH07]. The book contains many more technical aspects and only the relevant parts are included in this thesis. In addition, the book is a hands-on book which details how to set up and maintain honeypots.

The basic methodology introduced in Chapter 3 is based on joint work with Freiling and Wicherski [FHW05] on tracking botnet with a central server. A generalized version was developed in collaboration with Steiner, Dahl, Biersack, and Freiling [HSD+08].

The techniques and results presented in Chapter 4 were published in a preliminary form in several publications. Different techniques to utilize honeypots to capture malware samples in an automated way were published in a paper together with Bächer, Kötter, Freiling, and Dornseif [Hon05, BKH+06], and in joint work with Zhuge, Han, Song, and Zou [ZHH+07]. A method for automated malware analysis was published together with Willems and Freiling [WHF07]. Measurement results were published in joint work with Göbel and Willems [GHW07]. Finally, a system to detect IRC-based botnets was presented in joint work with Göbel [GH07].

A preliminary version of the work presented in Chapter 5 was published in a paper together with Steiner, Dahl, Biersack, and Freiling [HSD+08]. This thesis contains new insights and measurement results into the mechanisms behind peer-to-peer botnets.

Finally, Chapter 6 is based on joint work with Gorecki, Rieck, and Freiling [HGRF08] and Nazario [NH08] on fast-flux service networks.

The following publications were not included in this thesis since their focus was out of scope: together with Böhme, we studied the effects of stock spam on financial markets and we could show that spam messages can impact the traded volume and share price of stocks [BH06]. In joint work with Engelberth and Freiling, we studied in a related research project an active underground economy that trades stolen digital credentials [HEF08]. The results helps us better understand the nature and size of these quickly emerging underground marketplaces. Together with Ikinci and Freiling, we published a concept of a client-side honeypot, i.e., a type of honeypot that can be used to study attacks against client systems [IHF08]. A measurement study on malicious websites that attack web browsers was performed in collaboration with Zhuge, Song, Guo, Han, and Zou [ZHS+08]. Finally, together with Rieck, Willems, Düssel, Laskov, and Freiling we developed a system to classify malware samples based on their behavior observed in a controlled environment [RHW+08].

Chapter 2

Background and Related Work

2.1 Introduction

Before we can get started with a technical discussion of malicious remote control networks, some background information on *honeypots* is helpful since we use this concept as a basic building block throughout this thesis. A honeypot is a closely monitored computing resource that we want to be probed, attacked, or compromised. In a more precise way, we use the following definition throughout this work [Mai03]:

> A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource

We deliberately deploy systems that can be compromised by an attacker and use these decoy systems to learn more about actual attacks. A *honeynet* is a network of (possibly different kinds of) honeypots. In this chapter, we provide more details about the concept behind honeypots and honeynets. This discussion serves as a motivation for our work on malicious remote control networks.

Furthermore, we discuss work that is related to topics of this thesis in the second part of this chapter. This discussion provides an overview of previous work and highlights the contributions of this thesis. In later chapters we discuss related work in more detail where it is appropriate.

Outline. This chapter is outlined as follows: in Section 2.2 we provide an overview of the different concepts in the area of honeypots and honeynets. We describe the different approaches and discuss their advantages and limitations. Section 2.3 provides a brief overview of bots and botnets. Other work related to this thesis is discussed in Section 2.4 and we conclude this chapter with a summary in Section 2.5.

2.2 Honeypots and Honeynets

A honeypot is a closely monitored computing resource (e.g., a personal computer or a router) that we want to be probed, attacked, or compromised. The value of a honeypot is weighted by the information that can be obtained from it. Monitoring the data that enters and leaves a honeypot lets us gather information that is not available to common network-based intrusion detection systems (NIDS). For example, we can log the keystrokes of an interactive session even if encryption is used to protect the network traffic. To detect malicious behavior, NIDS typically requires signatures of known attacks and often fail to detect compromises that were unknown at the time it was deployed. On the other hand, honeypots can detect vulnerabilities that are not yet understood. For example, we can detect compromise by observing network traffic leaving the honeypot, even if the mechanism of the exploit has never been seen before. Because a honeypot has no production value, any attempt to contact it is suspicious by definition. Consequently, forensic analysis of data collected from honeypots is less likely to lead to false positives than data collected by NIDS. Most of the data collected with the help of a honeypot can help us to understand attacks.

Honeypots can run any operating system and any number of services. The configured services determine the vectors available to an adversary for compromising or probing the system. A *high-interaction honeypot* provides a real system the attacker can interact with. In contrast, a *low-interaction honeypot* simulates only some parts for example, the network stack [Pro04]. A high-interaction honeypot can be compromised completely, allowing an adversary to gain full access to the system and use it to launch further network attacks. In contrast, low-interaction honeypots simulate only services that cannot be exploited to get complete access to the honeypot. Low-interaction honeypots are more limited, but they are useful to gather information at a higher level — for example, to learn about network probes or worm activity. Neither of these two approaches is superior to the other; each has unique advantages and disadvantages that we discuss in this chapter and the rest of this thesis.

We also differentiate between *physical* and *virtual* honeypots. A physical honeypot is a real machine on the network with its own IP address. A virtual honeypot is simulated by another machine that responds to network traffic sent to the virtual honeypot. When gathering information about network attacks or probes, the number of deployed honeypots influences the amount and accuracy of the collected data. A good example is measuring the activity of HTTP-based worms [SMS01]. We can identify these worms only after they complete a TCP handshake and send their payload. However, most of their connection requests will go unanswered because they contact randomly chosen IP addresses. A honeypot can capture the worm payload by configuring it to function as a web server or by simulating vulnerable network services. The more honeypots we deploy, the more likely one of them is contacted by a worm.

In the following, we provide a more detailed overview of the individual types of honeypots and described their advantages and drawbacks.

2.2.1 High-Interaction Honeypots

A *high-interaction honeypot* is a conventional computer system, such as a commercial off-the-shelf (COTS) computer, a router, or a switch. This system has no conventional task in the network and no regularly active users. Thus, it should neither have any unusual processes nor generate any network traffic except regular daemons or services running on the system. These assumptions aid in attack detection: every interaction with the high-interaction honeypot is suspicious and could point to a possibly malicious action. This absence of false positives is one of the key advantages of honeypots compared to traditional intrusion detection systems (IDS). Hence, all network traffic going to and coming from the honeypot is logged. In addition, all system activity is recorded for later in-depth analysis.

It is also common to combine several honeypots to a network of honeypots, a *honeynet*. Usually, a honeynet consists of several honeypots of different type (different platforms and/or operating systems). This allows us to simultaneously collect data about different types of attacks. Usually we can learn in-depth information about attacks with the help of honeynets and therefore get qualitative results of attacker behavior, e.g., we can obtain a copy of the tools used by the attackers.

A honeynet creates a fishbowl environment that allows attackers to interact with the system while giving the operator the ability to capture all of their activity. This fishbowl also controls the attacker's actions, mitigating the risk of them damaging any non-honeypot systems. One key element to a honeynet deployment is called the *Honeywall*, a layer 2 bridging device that separates the honeynet from the rest of the network (see Figure 2.1 for a schematic overview of a high-interaction honeynet). This device mitigates risk through data control and captures data for analysis. Tools on the Honeywall allow for analysis of an attacker's activities. Any inbound or outbound traffic to the honeypots must pass through the Honeywall. Information is captured using a variety of methods, including passive network sniffers, IDS alerts, firewall logs, and the kernel module known as Sebek [The03]. The attacker's activities are controlled at the network level, with all outbound connections filtered through both an intrusion prevention system and a connection limiter.

Advantages and Disadvantages. With the help of a high-interaction honeypot, we can collect in-depth information about the procedures of an attacker. We can observe the "reconnaissance phase" — that is, how she searches for targets and with which techniques she tries to find out more about a given system. Afterward, we can watch how she attacks this system and which exploits she uses to compromise a machine. And finally, we can also follow her tracks on the honeypot itself: it is possible to monitor which tools she uses to escalate her privileges, how she communicates with other people, or the steps she takes to cover her tracks. Altogether, we learn more about the activities of an attacker — her tools, tactics, and motives. This methodology has proven to be successful in the past. For example, we were able to learn more about the typical procedures of phishing attacks and similar identity theft technique since we observed several of these attacks with the help of high-interaction honeypots [The05]. In addition,

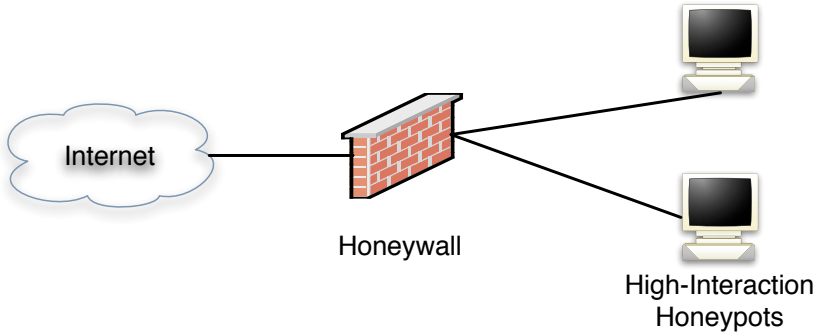


Figure 2.1: Example of honeynet environment with two high-interaction honeypots and a Honeywall.

honeypots have proven to be an efficient tool in learning more about Internet crime like credit card fraud [McC03a] or botnets [McC03b]. Several typical attacks we observed in the past are described in a book [PH07].

High-interaction honeypots — both virtual and physical — also bear some risks. In contrast to a low-interaction honeypot, the attacker can get full access to a conventional computer system and begin malicious actions. For example, she could try to attack other hosts on the Internet starting from the honeypot, or she could send spam from one of the compromised machines. This is the price we pay for gathering in-depth information about her procedures. However, there are ways to safeguard the high-interaction honeypots and mitigate this risk. The typical setup of a honeynet includes the Honeywall which filters known attack traffic and thus mitigates some of the risk. However, the Honeywall also does not offer complete protection since it cannot detect all kinds of attacks [The05].

One disadvantage of virtual honeypots is that the attacker can detect the monitoring system installed in a high-interaction honeypot [DHK04, HR05]. It might happen that an advanced attacker compromises a honeypot, detects the monitoring environment Sebek, and then leaves the suspected honeypot again. Moreover, she could change her tactics in other ways to try to fool the investigator. Therefore high-interaction honeypots could lead to less information about attackers.

Another drawback of high-interaction honeypots is the higher maintenance cost: we need to carefully monitor all honeypots and closely observe what is happening. Analyzing a compromise also takes some time. In our experience, analyzing a complete incident can take hours or even several days and one hour of attacker activity typically requires 10-20 hours of analysis [PH07].

2.2.2 Low-Interaction Honeypots

In contrast, *low-interaction honeypots* emulate services, network stacks, or other aspects of a real machine. They allow an attacker a limited interaction with the target system

and allow the operator to learn mainly quantitative information about attacks. For example, an emulated HTTP server could just respond to a request for one particular file and only implement a subset of the whole HTTP specification. The level of interaction should be "just enough" to trick an attacker or an automated tool, such as a worm that is looking for a specific file to compromise the server. Low-interaction honeypots can also be combined into a network, forming a *low-interaction honeynet*. The advantage of low-interaction honeypots is their simplicity and easy maintenance. Usually such a system can be deployed and it then collects data with low maintenance cost. The collected data could be information about propagating network worms [BKH+06] or scans caused by spammers for open network relays [Pro04]. Moreover, installation is generally easier for this kind of honeypot.

Low-interaction honeypots can primarily be used to gather statistical data and to collect high-level information about attack patterns [Hol06]. Furthermore, they can be used as a kind of intrusion detection system where they provide an early warning, i.e., a kind of burglar alarm, about new attacks [Pro04, GHH06], or they can be deployed to lure attackers away from production machines [Pro04, Sym, Sof]. In addition, low-interaction honeypots can be used to detect worms, distract adversaries, or to learn about ongoing network attacks. In Chapter 4, we introduce different types of low-interaction honeypots that can be used to collect information about malware that propagates in an autonomous manner.

An attacker cannot fully compromise the system since she interacts just with an emulation. Low-interaction honeypots construct a controlled environment and thus the risk involved is limited: the attacker should not be able to completely compromise the system, and thus the operator does not have to fear that she abuses the honeypots.

Advantages and Disadvantages. When an adversary exploits a high-interaction honeypot, she gains capabilities to install new software and modify the operating system. This is not the case with a low-interaction honeypot: a low-interaction honeypot provides only very limited access to the operating system. By design, it is not meant to represent a fully featured operating system and usually cannot be completely exploited. As a result, a low-interaction honeypot is not well suited for capturing *zero-day exploits*, i.e., attacks that exploit an unknown vulnerability or at least a vulnerability for which no patch is available. Instead, a low-interaction honeypot can be used to detect known exploits and measure how often a given network gets attacked. The term low-interaction implies that an adversary interacts only with an emulated environment that tries to deceive her to some degree, but does not constitute a fully fledged system. A low-interaction honeypot often simulates a limited number of network services and implements just enough of the Internet protocols, usually TCP and IP, to allow interaction with the adversary and make her believe she is connecting to a real system.

The advantages of low-interaction honeypots are manifold. They are easy to set up and maintain. They do not require significant computing resources, and they cannot be compromised by adversaries. The risk of running low-interaction honeypots is much smaller than running honeypots that adversaries can break into and control.

On the other hand, that is also one of the main disadvantages of the low-interaction honeypots. They only present the illusion of a machine, which may be pretty sophisticated, but it still does not provide an attacker with a real environment she can interact with. In general, it is often easy to fingerprint a low-interaction honeypot and detect its presence. An attacker could therefore scan the network for low-interaction honeypots in advance and not attack these machines.

Comparison of Both Approaches. Table 2.1 provides a summarized overview of high- and low-interaction honeypots, contrasting the important advantages and disadvantages of each approach.

	High-Interaction Honeypots	Low-Interaction Honeypots
Modus operandi	Real services, operating systems, or applications	Emulation of TCP/IP stack, vulnerabilities, ...
Involved risk	Higher risk	Lower risk
Maintenance	Hard to deploy and maintain	Easy to deploy and maintain
Captured data	Capture extensive amount of information	Capture quantitative information about attacks

Table 2.1: Comparison of advantages and disadvantages of high- and low-interaction honeypots

2.2.3 Physical Honeypots

Another possible distinction in the area of honeypots differentiates between *physical* and *virtual* honeypots. Physical honeypot means that the honeypot is running on a physical machine. Physical often implies high-interaction, thus allowing the system to be compromised completely. They are typically expensive to install and maintain. For large address spaces, it is impractical or impossible to deploy a physical honeypot for each IP address. In that case, the preferred approach is to use virtual honeypots.

2.2.4 Virtual Honeypots

The main advantages of virtual honeypots are scalability and ease of maintenance. We can deploy thousands of honeypots on just one physical machine. They are inexpensive to deploy and accessible to almost everyone.

Compared to physical honeypots, this approach is more lightweight. Instead of deploying a physical computer system that acts as a honeypot, we can also deploy one physical computer that hosts several virtual machines that act as honeypots. This leads to easier maintenance and lower physical requirements. Usually VMware [VMw], User-Mode Linux (UML) [UML], or other virtualization tools are used to set up such virtual honeypots. These tools allow us to run multiple operating systems and their

applications concurrently on a single physical machine, making it much easier to collect data. Moreover, other types of virtual honeypots are introduced in Chapter 4, in which we focus on malicious remote control networks with a central server.

To start implementing the high-interaction methodology, a user can simply use a physical machine and set up a honeypot on it. However, choosing an approach that uses virtual high-interaction honeypots is also possible: instead of deploying a physical computer system that acts as a honeypot, the user can deploy one physical computer that hosts several virtual machines that act as honeypots. This has some interesting properties. First, the deployment is not very difficult. There are some solutions that offer an already pre-configured honeypot that just needs to be customized to the local environment and can then be executed. Second, it is the easy maintenance. If an attacker compromises a honeypot, the operator can watch her and follow her movements. After a certain amount of time, the operator can restore the honeypot to the original state within minutes and start from the beginning. Third, using a virtual machine to set up a honeypot poses less risk because an intruder is less likely to compromise or corrupt the actual machine on which the system is running.

One disadvantage of virtual honeypots is that the attacker can differentiate between a virtual machine and a real one [HR05]. It might happen that an advanced attacker compromises a virtual honeypot, detects the virtual environment, and then leaves the honeypot again without performing any attacks. This is similar to the drawbacks of high-interaction honeypots introduced above.

2.2.5 Honeyclients

Honeyclients, which are sometimes also denominated as *client-side honeypots*, are the opposite to the "classical" server honeypot solutions. The main idea is to emulate the behavior of humans and then closely observe whether or not the honeypot is attacked [WBJ+06]. For example, a honeyclient can actively surf websites to search for malicious web servers that exploit the visitor's web browser (so called *drive-by download attacks* [PMRM08]) and thus gather information of how attackers exploit client systems. Another example are honeyclients that automatically process e-mails by clicking on attachments or following links from the e-mail and then closely observe if the honeypot is attacked. The current focus in the area of honeyclients is commonly based on the analysis of web client exploitation, since this attack vector is often used by attackers in order to compromise a client.

Honeyclients also have another advantage: one major culprit of server honeypots is their waiting for an attack — they are similar to a burglar alarm, waiting for an incident to happen. It is possible that a ready to be attacked honeypot will not be attacked for weeks or even months, or it is also possible that the honeypot is attacked occasionally by many attackers at the same time. In general, it is thus not predictable how frequently attacks will occur on a honeypot and therefore the analyses get more complicated. In comparison, honeyclients initialize the analysis phase by visiting websites and thus control the maximum number of possible attacks. Furthermore, the operator of a

honeyclient can run several clients in parallel and scale how much information is processed by them.

Honeyclients can also be classified as high-interaction or low-interaction honeyclients. High-interaction honeyclients are usually real, automated web browsers on real operating systems which interact with websites like real humans would do. They log as much data as possible during the attack and usually allow a fixed time period for an attack. Since they provide detailed information about the attack, high-interaction honeyclients are in general rather slow and not able to scan broad parts of the web. Low-interaction honeyclients, on the other hand, are often emulated web browsers, usually web crawlers, which do have no or only limited abilities for attackers to interact with. Low-interaction honeyclients often make use of static signature or heuristics-based malware and attack detection tools and thus lack the detection of zero-day exploits and unimplemented attack types. These honeyclients are not suited for an in-depth investigation of an attacker's actions after a successful compromise because the system is only simulated and any other action than the initial exploitation is likely to be missed, too. In spite of these drawbacks, low-interaction honeyclients are often easy to deploy and operate and usually have a high performance: they can be used for analyzing a large number of URLs compared to the high-interaction variant. Furthermore, the containment of attacks can be implemented in a straightforward way because the compromised system is not real and thus unusable for the attacker, which simplifies the deployment of a low-interaction honeyclient and minimizes the risk involved.

2.3 Bots and Botnets

In this section, we present a brief overview of bots and botnets to provide a first glance at the malicious remote networks we cover in this thesis. Chapter 4 and Chapter 5 provides a more detailed and extensive overview of different aspects of botnets.

The term *bot* is derived from the word *robot* and refers to a program which can, to some degree, act in an autonomous manner. A computer system that can be remotely controlled by an attacker is called a *bot* or *zombie*. Bots started off as little helper tools, especially in the IRC community, to keep control of a private channel or as a quiz robot, randomly posting questions. In the context of malware, bots are harmful programs, designed to do damage to other hosts on the network [Hol05]. Moreover, bots can be grouped to form so called *botnets*, consisting of several hundreds up to thousands of hosts, whose combined power can be utilized by the owner of the botnet to perform powerful attacks. One of these powerful attacks are *Distributed Denial of Service* (DDoS) attacks, which overwhelm the victim with a large number of service requests. DDoS attacks are described in more detail in the work by Mirkovic and Reiher [MR04]. Other abuses of bots can be identity theft, sending of spam emails and similar nefarious purposes (see Chapter 4 and Chapter 5 for details).

To control a large group of bots, a *Command and Control* (C&C) server is utilized, which all zombie machines connect to and receive instructions from. Figure 2.2 provides a schematic overview of a botnet. A common method of an attacker to communicate

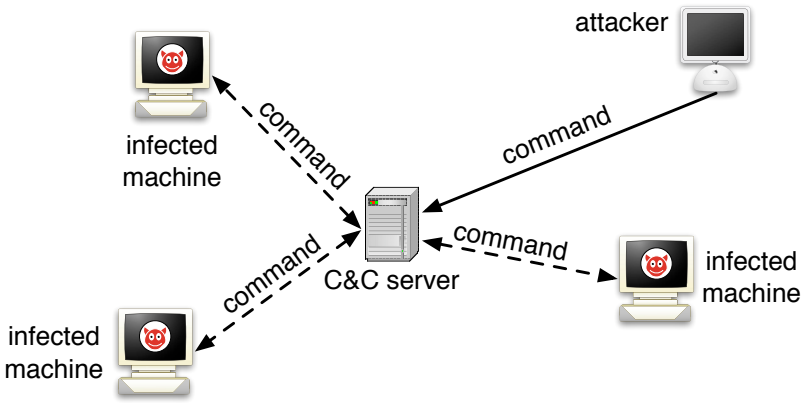


Figure 2.2: Setup of a botnet with a central server for Command & Control.

with the botnet is to use the IRC protocol as communication channel. In this case, the infected machines automatically join a specific channel on a public or private IRC server to receive further instructions. This could for example be the command to perform an attack against a specified victim or to scan certain network ranges for hosts with known vulnerabilities. It is not even necessary for a bot to join a channel, there are also bots which use private messages to receive instructions from the botnet owner. Thus, a connection to the C&C server suffices. Besides IRC, other protocols are used more and more by bots. For example, HTTP is nowadays a common communication channel. In this case, the bot periodically polls the C&C server and interprets the responses as commands. Several bots also use peer-to-peer based protocols to avoid a central C&C server and this form of communication could become more prevalent in the future. We focus on these botnets in Chapter 5.

Since bots are able to autonomously propagate further across the network and feature keylogging and backdoor functionalities as well, they can be seen as a combination of worms, rootkits and Trojan horses. A more detailed technical description of botnets and their attack features is provided in a paper we published under the name of the Honeynet Project [Hon05].

2.4 Related Work

We now discuss the related work in the areas of honeypots/honeynets and bots/botnets. This discussion provides an overview of work related to the methodology presented in this thesis. In later sections, we discuss related work in more detail where it is appropriate. Our work also touches on the areas of network- and host-based security, but since this is not the focus of this thesis, we refrain from a detailed discussion of related work in that area here.

2.4.1 Honeypots and Honeynets

The area of honeypot-related research has received a lot of attention in the recent years with many new concepts and tools being developed [BCJ+05, MSVS04, JX04, LD08, PSB06, Pro04, VBBC04, VMC+05, YBP04]. All relevant concepts were already discussed in Section 2.2 and thus we focus on a few important systems here. Low-interaction honeypots related to our work are discussed in more detail in Section 4.4.

The most common setup for high-interaction honeynets are so called *GenIII honeynets* [BV05]. In such a setup, several high-interaction honeypots are equipped with the monitoring software *Sebek* [The03] which permanently collects information about the system state. All logged information is sent to the *Honeywall*, that collects this information in a database. Furthermore, the Honeywall is responsible for *Data Control*, i.e., control suspicious traffic entering or leaving the honeynet, and *Data Analysis*, i.e., it provides an analysis frontend for the operator. We use GenIII honeynets throughout this thesis for either capturing or analyzing malicious software.

Honeyd is a honeypot framework developed by Provos to instrument thousands of Internet addresses with virtual honeypots and corresponding network services [Pro04]. Each IP address can be individually configured to simulate a specific configuration. Due to the scalability of the tool, it is applicable to very large honeynet setups. Honeyd inspired the work presented in this thesis. We discuss the relationship between Honeyd and the low-interaction honeypots that emulate vulnerable network services in more detail in Section 4.4.

The HoneyMonkey project is a web browser (Internet Explorer) based high-interaction honeyclient developed by Wang et al. [WBJ+06]. The architecture consists of a chain of virtual machines with different flavors of the Windows Operating system in various patch levels. Starting with a fully unpatched system, the Monkey Controller initiates a so called "monkey" program which browses previously scheduled web sites. The monkey opens the web site and waits for a predefined time. After the time-out, the virtual machine is checked for signs of a successful intrusion. If an attack is detected, the web site is revisited with the next machine having a higher patch-level in the pipeline. This analysis process is repeated as long as the machines are successfully compromised by visiting a particular web site. If finally the last, fully patched system in the chain is also exploited, then the system has found a zero-day vulnerability.

The MITRE honeyclient project is an open-source, web browser based high-interaction honeyclient developed by Wang et al. [Wan09]. Another similar tool is Capture-HPC developed by Seifert and Steenson [SS09]. Both tool are high-interaction honeyclients and follow the same basic mechanisms as HoneyMonkey. For our work in this thesis, we use honeyclients to detect malicious websites and collect samples of malicious software that propagates using this attack vector.

2.4.2 Bots and Botnets

In the last few years, the botnet phenomenon got the general attention of the security research community. We published one of the first systematic studies of botnets under

the name of the Honeynet Project [Hon05]. This work provides a detailed overview of the technical aspects of bots and botnets and served as one of the foundations of this thesis. Cooke et al. [CJM05] outlined the origins and structure of botnets and present some results based on a distributed network sensor and honeypots. However, they do not give detailed results that characterize the extent of the botnet problem. Rajab et al. [RZMT06] used DNS data and monitoring of C&C control activity to get a better understanding of botnets. Their study was based on data collected by monitoring 192 botnets during a period of more than three months. They use low-interaction honeypots that emulate vulnerabilities in network services as we describe in Section 4.4. Dagon et al. [DZL06] studied the diurnal patterns in botnets and proposed using DNS sinkholing to study botnets. Botnets that use peer-to-peer based protocols in the wild were first described in a methodical way by Grizzard et al. [GSN+07].

Canavan [Can05] and Barford and Yegneswaran [BY07] presented an alternative perspective on IRC-based botnets based on in-depth analysis of the source code from several bot families. We also analyzed the source code of several bot families such as SdBot and Agobot, which can be freely downloaded from the Internet, to understand some of the effects we monitored in the wild.

A lot of systems were presented to detect botnets and we just provide an overview of the most important systems and techniques proposed in the literature. Gu introduced many different tools like BotHunter [GPY+07], BotSniffer [GZL08], and BotMiner [GPZL08] that detect different aspects of botnet behavior and the communication channel used by bots. The results of this work are summarized in his PhD thesis [Gu08]. A similar approach to BotMiner was presented by Reiter and Yen [YR08]. All these tools try to find either *horizontal* or *vertical* correlations in network communication. A horizontal correlation implies that several host behave similar at the network level: this indicates infected machines under a common control infrastructure which respond to a given command. The second approach, namely vertical correlation, tries to detect individual machines that behave like infected machines, for example by detecting typical network signatures of botnet communication. In this thesis, we do not focus on botnet detection, but present a methodology to stop this kind of malicious remote control networks in a general way.

A transport layer-based botnet detection approach was introduced by Karasaridis et al. [KRH07]. They use passive analysis based on flow data to characterize botnets and were able to detect several hundred controllers over a period of seven months. However, such a flow-based approach cannot provide insight into the botnet and the control structure itself. With the method proposed in this thesis, we can observe the commands issued by an attacker, the executables used, and similar effects of a botnet.

Botnets commonly use the same IRC channel for bots. This observation is used by Binkley and Singh to detect suspicious IRC servers [BS06, Bin06]. They combine TCP-based anomaly detection with IRC-based protocol analysis and are able to detect botnets efficiently. The system collects statistics over a complete day and aggregates the collected information. Chen presented a system that tries to detect botnet traffic at edge network routers and gateways [Che06]. In addition, this work includes preliminary statistics like mean packet length of IRC packets and the distribution of IRC messages

like JOIN or PING/PONG, but it does not give statistics about the success rate of the approach. Strayer et al. use a similar approach to examine flow characteristics such as bandwidth, duration, and packet timing [SWLL06].

2.5 Summary

In this chapter, we introduced the concept of honeypots. These systems are conventional computer systems like an ordinary PC system or a router, deployed to be probed, attacked, and compromised. Honeypots are equipped with additional software that constantly collects information about all kinds of system activity. This data greatly aids in post-incident computer and network forensics. In the rest of this thesis, we use the concept of honeypots as a basic block for our methodology.

We also discussed botnets, networks of compromised machines under the control of an attacker. Due to their sheer size and the harm then can cause, botnets are nowadays one of the most severe threats on the Internet. Botnets are an example of malicious remote control networks and we introduce in the next chapter a general root cause methodology to prevent them.

Chapter 3

Root-Cause Methodology to Prevent Malicious Remote Control Networks

3.1 Introduction

In this chapter, we introduce a general root cause methodology to prevent malicious remote control networks. This methodology uses honeypot-based techniques and is a novel application of honeynets. The basic idea of the approach is to first infiltrate the network, then analyze the network from the inside, and finally use the collected information to stop it. The presented methodology is one of the first preventive techniques that aims at DDoS attack avoidance, i.e., ensuring that DDoS attacks are stopped before they are even launched. We present an effective approach to DDoS prevention that neither implies a resource arms race nor needs any additional (authentication) infrastructure. Furthermore, this methodology can be used to stop many other threats caused by networks of compromised machines, e.g., spamming and identity theft. In the next three chapters, we exemplify this approach and show how we can use honeynets to prevent different kinds of attacks caused by botnets and similar malicious remote control networks.

Contributions. The contributions of this chapter are two-fold:

- We analyze the basic principles behind malicious remote control networks and provide key insights about their structure and configuration.
- Based on these insights, we introduce a general root cause methodology to prevent malicious remote control networks. This is achieved by capturing a sample related to the network, analyzing it, infiltrating the network with an agent, and then observing the network in detail. All these steps are automated to a high degree. Based on the collected information, the network can then be mitigated.

Outline. This chapter is outlined as follows: in the first part, we analyze the basic principles behind malicious remote control networks and substantiate our findings. In the second part, we propose a root cause methodology to prevent DDoS attacks. This methodology is rather general and allows us to prevent many different kinds of malicious remote control networks. In the next three chapters, we then exemplify the method with the help of several concrete implementations.

3.2 Methodology

In this section we introduce a general methodology to prevent DDoS attacks. This methodology can then be generalized to stop other kinds of threats that are based on malicious remote control networks as we show in later chapters. Our methodology is based on the following line of reasoning:

1. To mount a successful DDoS attack, a large number of compromised machines is necessary.
2. To coordinate a large number of machines, the attacker needs a remote control mechanism.
3. If the remote control mechanism is disabled, the DoS attack is prevented.

We will substantiate this line of reasoning in the following paragraphs.

3.2.1 A Large Number of Machines is Necessary

Why does an attacker need a large number of machines to mount a successful DDoS attack? If an attacker controls only one or a few machines, a DDoS attack is successful only if the total resources of the attacker (e.g., available bandwidth or possibility to generate many packets per second) are greater than the resources of the victim. Otherwise the victim is able to cope with the attack. Hence, if this requirement is met, the attacker can efficiently overwhelm the services offered by the victim or cause the loss of network connectivity.

Moreover, if only a small number of attacking machines are involved in an attack, these machines can be identified and counteractive measures can be applied, e.g., shutting down the attacking machines or blocking their traffic. To obfuscate the real address of the attacking machines, the technique of *IP spoofing*, i.e., sending IP packets with a counterfeited sender address, is often used. Furthermore, this technique is used to disguise the actual number of attacking machines by seemingly increasing it. However, IP spoofing does not help an attacker to conduct a DDoS attack from an efficiency point of view: it does not increase the available resources, but it even reduces them due to computing efforts for counterfeiting the IP addresses. In addition, several ways to detect and counteract spoofed sender address exist, e.g., ingress filtering [Fer00], packet marking [SP01], or ICMP traceback [SWKA00, Bel01]. The IP address distribution of a large number of machines in different networks makes ingress filter construction,

maintenance, and deployment much more difficult. Additionally, incident response is hampered by a high number of separate organizations involved.

Therefore control over a large number of machines is necessary (and desirable from an attacker's point of view) for a successful DDoS attack.

3.2.2 A Remote Control Mechanism is Necessary

The success of a DDoS attack depends on the volume of the malicious traffic as well as the time this traffic is directed against the victim. Therefore, it is vital that the actions of the many hosts which participate in the attack are well-coordinated regarding the type of traffic, the victim's identity, as well as the time of attack.

A cautious attacker may encode all this information directly into the malware which is used to compromise the zombies that form the DDoS network. While this makes her harder to track down, the attacker loses a lot of flexibility since she needs to plan her deeds well in advance. Additionally, this approach makes the DDoS attack also less effective since it is possible to analyze the malware and then reliably predict when and where an attack will take place. Therefore it is desirable for an attacker to have a channel through which this information can be transferred to the zombies on demand, i.e., a remote control mechanism.

A remote control mechanism has several additional advantages:

1. The most effective attacks come by surprise regarding the time, the type, and the target of attack. A remote control mechanism allows an attacker to react swiftly to a given situation, e.g., to mount a counterattack or to substantiate blackmail threats.

2. Like any software, malware is usually far from perfect. A remote control mechanism can be used as an automated update facility, e.g., to upgrade malware with new functionality or to install a new version of the malware once antivirus engines start to detect the malware.

In short, a DDoS attack mechanism is more effective if an attacker has some type of remote control over a large number of machines. Then she can issue commands to exhaust the victim's resources at many systems, thus successfully attacking the victim.

3.2.3 Preventing Attacks

Our methodology to mitigate DDoS attacks aims at manipulating the root cause of the attacks, i.e., influencing the remote control network. Our approach is based on three distinct steps:

1. Infiltrating the remote control network.

2. Analyzing the network in detail from the inside.

3. Shutting down the remote control network.

In the first step, we have to find a way to smuggle an *agent* into the control network. In this context, the term agent describes a general procedure to mask as a valid member of the control network. This agent must thus be customized to the type of network we want to plant it in. The level of adaptation to a real member of the network depends on the target we want to infiltrate. For instance, to infiltrate a botnet we would try to simulate a valid bot, maybe even emulating some bot commands. We could also run a copy of the actual bot in a controlled environment and closely monitor what the bot is doing, thereby passively infiltrating the remote control network.

Once we are able to sneak an agent into the remote control network, it enables us to perform the second step, i.e., to observe the network in detail. So we can start to monitor all activity and analyze all information we have collected. For example, after having achieved to smuggle an agent into a botnet, we start to monitor all commands issued by the attacker and we can also observe other victims joining and leaving the botnet command channel.

In the last step, we use the collected information to shut down the remote control network. Once this is done, we have deprived the attacker's control over the other machines and thus efficiently stopped the threat of a DDoS attack with this network. Again, the particular way in which the network is shut down depends on the type of network. For example, to shut down an IRC-based botnet that uses a central server to distribute the attacker's commands, we could use the following approach: since the C&C server is the only way for an attacker to issue commands to all bots, a viable approach to shut down the network is to disconnect the C&C server from the Internet. This would stop the flow of commands from the attacker to the bots, effectively stopping the whole botnet. For botnets that use peer-to-peer protocols for communication, the shutdown is harder as we exemplify in Chapter 5. And we can also indirectly stop the malicious remote control network by interfering with essential communication mechanisms like DNS as we show in Chapter 6.

3.2.4 Discussion

The methodology described above can be applied to different kinds of malicious remote control networks and is thus very general. The practical challenge of the methodology is to *automate* the infiltration and analysis process as much as possible. If it is possible to "catch" a copy of the malware in a controlled way, we can set up an automated system that "collects" malware. In this thesis, we show how the idea behind honeypots can be used to construct such a system. In all cases, the malware needs to establish a communication channel between itself and the attacker to receive commands. We can exploit this fact to extract a lot of information out of the communication channel in an automated fashion. For example, if contact to the attacker is set up by establishing a regular network connection, the network address of the attacker's computer can be automatically determined. By implementing more general *behavior-based* analysis techniques, we can extract even more information from a given malware sample that can then be used for the infiltration and analysis process.

To many readers, the methodology may sound like coming directly from a James Bond novel and it is legitimate to ask for evidence of its feasibility. In the following chapters we give exactly this evidence. We show that this method can be realized on the Internet by describing how we infiltrated and tracked different kinds of malicious remote control networks. We focus on different kinds of networks:

1. *IRC-based botnets*: these are the typical botnets seen in the wild nowadays. They use IRC for command and control and we show an efficient mechanism to learn more about this threat in Chapter 4.

2. *HTTP-based botnets*: this kind of botnets uses HTTP for command and control. In contrast to IRC-based botnets, the individual bots periodically query the C&C server for new commands and thus this communication model implements a *polling* mechanism. We study these networks also in Chapter 4.

3. *Peer-to-Peer-based botnets*: instead of using a central server for command and control, these botnets use peer-to-peer protocols for communication. It is nevertheless possible to track these networks, but shutting them down is harder as we illustrate in Chapter 5.

4. *Fast-flux service networks*: the attackers use the compromised machines to establish a proxy network on top of these machines to implement a robust hosting infrastructure. The same basic idea can also be used to study this threat as we demonstrate in Chapter 6.

With the help of these four examples, we substantiate our approach and show in the following, that a high degree of automation is possible and practical. In addition, our approach does not need any secrecy for the involved procedures, we can publish all details. If the communication flow of the remote control network changes, we need to slightly adjust our algorithms, but the general approach stays the same.

3.3 Summary

DDoS attacks have become increasingly dangerous in recent years and we are observing a growing professionalism in the type of Internet crime surrounding DDoS. In this chapter we have introduced a technique for DDoS attack prevention that neither implies a resource arms race nor needs any additional infrastructure. In contrast to previous work in this area our approach is preventive instead of reactive. Our technique attacks a root cause of DDoS attacks: in order to be effective, an attacker has to control a large number of machines and thus needs a remote control network. Our methodology aims at shutting down this control network by infiltrating it and analyzing it in detail. Besides DDoS attacks, our approach can also be used to prevent other illicit uses of malicious remote control networks like spamming or mass identity theft since we stop the control mechanisms behind of all these threats. In the next three chapters, we show the practical feasibility of the proposed methodology.

Chapter 4

Tracking Botnets with Central C&C Server

4.1 Introduction

In this chapter we exemplify a technical realization of the methodology we introduced in the previous chapter. We present an approach to track and observe botnets that is able to prevent DDoS attacks. Since we stop the whole botnet operation, also additional threats like sending of spam mails, click fraud, or similar abuses of botnets are stopped. In this chapter, we focus on malicious remote control networks with a *central* Command & Control (C&C) server, botnets with other communication structures are covered in the following chapters.

As already stated previously, tracking botnets is a multi-step procedure: first we need to gather some data about an existing botnet. This can for instance be obtained with the help of honeypots and via an analysis of captured malware. We show in this chapter how both steps can be automated to a high degree. With the help of this information it is possible to smuggle an agent into the network as we show afterwards.

Contributions. The contributions of this chapter are threefold:

1. We show how the method of tracking malicious remote control networks as introduced in Chapter 3 can be used to track botnets with a central C&C server. We argue that the method is applicable to analyze and mitigate *any* botnet using a central control structure.

2. We demonstrate the applicability by performing a case study of IRC- and HTTP-based botnets, thereby showing how different kinds of botnets can be tracked with our methodology.

3. We present empirical measurement results obtained by applying the methodology in real-world settings on the Internet.

Outline. This chapter is outlined as follows: in Section 4.2 we motivate our work on preventing botnets since these networks are the root cause behind DDoS and other attacks and our method is one of the first preventive methods to stop this threat. Section 4.3 provides a detailed overview of bots and botnets with a central C&C server, including technical details behind these networks. We explain how samples of these bots can be collected in an automated way (Section 4.4), how these samples can be automatically analyzed (Section 4.5), and how we can infiltrate them (Section 4.6). A realization of the methodology proposed in Chapter 3 to track botnets with an IRC C&C server is presented in Section 4.7.1. The same methodology also applies to HTTP-based botnets as we show in Section 4.7.2. Empirical measurement results are presented in Section 4.8 and we briefly discuss mitigation strategies in Section 4.9. Finally, we conclude in Section 4.10 with a summary.

4.2 Motivation

As discussed in previous chapters, the root cause behind DDoS attacks nowadays are botnets, i.e., networks of compromised machines under the control of an attacker. Defensive measures against DDoS attacks can be classified as either preventive or reactive [MR04]. Currently, reactive techniques dominate the arena of DDoS defense methods (the work by Mirkovic *et al.* [MDDR04] gives an excellent survey over academic and commercial systems). The idea of reactive approaches is to detect the attack by using some form of (distributed) anomaly detection on the network traffic and then react to the attack by reducing the malicious network flows to manageable levels [MRRK03]. The drawback of these approaches is that they need an increasingly complex and powerful sensing and analysis infrastructure to be effective: the approach is best if large portions of network traffic can be observed for analysis, preferably in real-time.

Preventive methods either eliminate the possibility of a DDoS attack altogether or they help victims to survive an attack better by increasing the resources of the victim in relation to those of the attacker, e.g., by introducing some form of strong authentication before any network interaction can take place (see for example work by Meadows [Mea98]). Although being effective in theory, these survival methods always boil down to an arms race between attacker and victim where the party with more resources wins. In practice, it seems as if the arms race is always won by the attacker, since it is usually easier for her to increase her resources (by compromising more machines) than for the victim, which needs to invest money in equipment and network bandwidth.

Preventive techniques that aim at DDoS attack avoidance (i.e., ensuring that DDoS attacks are stopped before they are even launched) have received close to no attention so far. One reason for this might be the popular folklore that the only effective prevention technique for DDoS means to fix all vulnerabilities in all Internet hosts that can be misused for an attack (see for example Section 5 of the article by Mirkovic and Reiher [MR04]). In this thesis we show that this folklore is wrong by presenting an effective approach to DDoS prevention that neither implies a resource arms race nor

needs any additional (authentication) infrastructure. The approach is based on the observation that coordinated automated activity by many hosts is at the core of DDoS attacks. Hence the attacker needs a mechanism to remotely control a large number of machines. To prevent DDoS attacks, our approach attempts to identify, infiltrate and analyze this remote control mechanism and to stop it in an automated and controlled fashion. We have outlined this methodology already in detail in the previous chapter. Since we attack the problem of DDoS at the root of its emergence, we consider our approach to be a root cause method to DDoS defense. Furthermore, our approach also helps to mitigate other malicious activities caused by these remote control mechanisms like sending of spam mails or click fraud.

It may seem unlikely that it is possible to automatically analyze and infiltrate a malicious remote control method crafted by attackers for evil purposes. However, we provide evidence of the feasibility of our strategy by describing how we successfully tracked and investigated the automated attack activity of botnets on the Internet. The idea of our methods is to "collect" malware using honeypots, i.e., network resources (computers, routers, switches, etc.) deployed to be probed, attacked, and compromised (see Section 2.2 for details). Based on this idea, we can implement a system to automatically collect samples of autonomous spreading malware. From the automated analysis we derive the important information necessary to observe and combat malicious actions of the botnet maintainers. In a sense, our approach can be characterized as turning the methods of the attackers against themselves.

4.3 Technical Background

During the last few years, we have seen a shift in how systems are being attacked. After a successful compromise, a *bot* (often also referred to as *zombie* or *drone*) is commonly installed on the victim's system. This program provides a remote control mechanism to command this machine. Via this remote control mechanism, the attacker can issue arbitrary commands and thus has complete control over the victim's computer system. In Chapter 2, we provide more details about the mechanism behind these so called *botnets*. With the help of a botnet, attackers can control several hundred or even thousands of bots at the same time, thus enhancing the effectiveness of their attack. In this section we discuss concepts behind bots and botnets and introduce in detail the underlying methods and tools used by the attackers.

4.3.1 Overview of Bots

Historically, the first bots were programs used in Internet Relay Chat (IRC, defined in RFC 2810) networks. IRC was developed in the late 1980s and allows users to talk to each other in so-called IRC channels in real time. Bots offered services to other users, for example, simple games or message services. But malicious behavior evolved soon and resulted in the so-called *IRC wars*, one of the first documented DDoS attacks.

Nowadays, the term *bot* describes a remote control program loaded on a computer, usually after a successful invasion, that is often used for nefarious purposes. During the last few years, bots like Agobot, SDBot, RBot, and many others, were often used in attacks against computer systems. Moreover, several bots can be combined into a *botnet*, a network of compromised machines that can be remotely controlled by the attacker. Botnets in particular pose a severe threat to the Internet community, since they enable an attacker to control a large number of machines.

Three attributes characterize a bot: a remote control facility, the implementation of several commands, and a spreading mechanism to propagate it further:

- A remote control lets an attacker manipulate infected machines. Bots currently implement several different approaches for this mechanism:
  - Typically, the controller of the bots uses a central IRC-based C&C server. All bots join a specific channel on this server and interpret all the messages they receive there as commands. This structure is usually secured with the help of passwords to connect to the server, join a specific channel, or issue commands. Several bots also use SSL-encrypted communication.
  - In other situations, such as when some bots avoid IRC and use covert communication channels, the controller uses, for example, communication channels via HTTP or DNS instead of the IRC protocol. They can, for example, encode commands to the bots inside HTTP requests or within DNS TXT records. Another possibility is to hide commands in images (*steganography*).
  - Some bots use peer-to-peer communication mechanisms to avoid a central C&C server because it is a single point of failure. Chapter 5 discusses these botnets in greater detail.
- Typically, two types of commands are implemented over the remote control network: DDoS attacks and updates. DDoS attacks include SYN and UDP flooding or more clever ones such as spidering attacks — those that start from a given URL and follows all links in a recursive way — against websites. Update commands instruct the bot to download a file from the Internet and execute it. This lets the attacker issue arbitrary commands on the victim's machine and dynamically enhance the bot's features. Other commands include functions for sending spam, stealing sensitive information from the victim (such as passwords or cookies), or using the victim's computer for other nefarious purposes.

  The remote control facility and the commands that can be executed from it differentiate a bot from a *worm*, a program that propagates itself by attacking other systems and copying itself to them.
- But like a worm, most bots also include a mechanism to spread further, usually by automatically scanning whole network ranges and propagating themselves via vulnerabilities. These vulnerabilities usually appear in the Windows operating system, the most common being DCOM [Mic03], LSASS [Mic04], or one of the newer Microsoft security bulletins.

Attackers also integrate recently published exploits into their bots to react quickly to new trends. Propagation via network shares and weak passwords on other machines is another common technique: the bot uses a list of passwords and usernames to log on to remote shares and then drops its copy. Propagation as an e-mail attachment, similar to e-mail worms, can also be used as a propagation vector. Some bots propagate by using peer-to-peer filesharing protocols [SJB06, KAG06]. Using interesting filenames, the bot drops copies of itself into these program's shared folders. It generates the filename by randomly choosing from sets of strings and hopes that an innocent user downloads and executes this file.

An additional characteristic applies to most bots we have captured in the wild: most of them have at least one executable packer, a small program that compresses/encrypts the actual binary. Typically, the attacker uses tools such as UPX [OMR08], ASPack [Sta08] or PECompact [Tec08] to pack the executable. The packing hampers analysis and makes reverse engineering of the malware binary harder.

4.3.2 Examples of Bots

We now take a closer look at some specific bot variants to provide an overview of what type of bots can be found in the wild.

Agobot and Variants. One of the best-known families of bots includes Agobot/ Gaobot, its variants Phatbot and Forbot, and several others. Most antivirus vendors have hundreds of signatures for this family of bots. The source code for Agobot was published at various websites in April 2004, resulting in many new variants being created over time. The original version of Agobot was written by a young German, who was arrested and charged under the computer sabotage law in May 2004 [Uni07, Tho08]. The actual development team behind the bot consisted of more people. The bot is written in C++ with cross-platform capabilities and shows a very high abstract design.

For remote control, this family of bots typically uses a central IRC server for Command and Control. Some variants also use peer-to-peer communication via the decentralized WASTE network [Tea08], thus avoiding a central server. In the variant we have analyzed, eight DoS-related functions were implemented and six different update mechanisms. Moreover, at least ten mechanisms to spread further exist. This malware is also capable of terminating processes that belong to antivirus and monitoring applications. In addition, some variants modify the hosts file, which contains the host name to IP address mappings. The malware appends a list of website addresses – for example, of antivirus vendors – and redirects them to the loopback address. This prevents the infected user from accessing the specified location.

Upon startup, the program attempts to run a speed test for Internet connectivity. By accessing several servers and sending data to them, this bot tries to estimate the available bandwidth of the victim. This activity of the bot allows us to estimate the actual number of hosts compromised by this particular bot. This works by taking a look at log files – for example, Agobot uses `www.belwue.de` as one of the domains for this

speed test. So the administrators of this domain can make an educated guess about the actual number of infected machines by looking at how often this speed test was performed. In May 2004, about 300,000 unique IP addresses could be identified in this way per day [Fis05]. A detailed analysis of this bot was published by Stewart [Gro04b].

SDBot and Variants. SDBot and its variants RBot, UrBot, UrXBot, Spybot, are at the time of writing one of the most active bots in the wild. The whole family of SDBots is written in C and literally thousands of different versions exist, since the source code is publicly available. It offers similar features as Agobot, although the command set is not as large nor the implementation as sophisticated.

For remote control, this bot typically only offers the usage of a central IRC server. But there are also variants that used HTTP to command the bots. Again, the typical commands for remote control are implemented. More than ten DDoS attacks and four update functions were implemented in the bots we have analyzed. Moreover, this bot incorporates many different techniques to propagate further. Similar to Agobot and its variants, the whole family of SDBots includes more than ten different possibilities to spread further, including exploit to compromise remote systems and propagation with other mechanisms.

The evolution of bots through time can be observed by means of this family of bots. Each new version integrates some new features, and each new variant results in some major enhancements. New vulnerabilities are integrated in a couple of days after public announcement, and once one version has new spreading capabilities, all others integrate it very fast. In addition, small modifications exist that implement specific features (e.g., encryption of passwords within the malware binary) that can be integrated in other variants in a short amount of time.

Zotob/Mytob. Strictly speaking, Zotob is just a variant of Rbot. It gained much media attention, since it affected machines from companies such as CNN, ABC, and the New York Times. Zotob was one of the first bots to include the exploit for the Microsoft security bulletin MS05-039 [Mic05], released on August 9, 2005. Only four days after the security bulletin, Zotob began spreading and compromised unpatched machines. It spread quickly and during a live show, reporters from CNN reported that their computers were affected by a new worm. But the fame for the botmasters was short-lived: 12 days after the first release, on August 25, the Moroccan police arrested, at the request of the FBI, two suspected botmasters [Fed06]. At the same time, another young man from Turkey was arrested as another suspect in this case.

Storm Worm. Starting with Nugache [Naz06], we have seen more and more bots that use peer-to-peer-based protocols for botnet command and control. One prominent example is Storm Worm, for which a detailed analysis is available by Stewart [Ste07]. This particular piece of malware uses a variation of the eDonkey protocol to exchange command and update messages between the bots. Storm Worm was mainly used to send spam mails and to attack a number of antispam websites via DDoS attacks.

Since this botnet does not have a central server used for C&C, it is rather hard to track it, and shutting it down is even harder. In the next chapter, we show how the methodology introduced in Chapter 3 can be used to track botnets with a peer-to-peer-based communication channel and we exemplify this technique with a case study on Storm Worm, the most prevalent botnet observed so far.

Bobax. An interesting approach in the area of bots is Bobax. It uses HTTP requests as communication channel and thus implements a stealthier remote control than IRC-based C&C. In addition, it implements mechanisms to spread further and to download and execute arbitrary files. In contrast to other bots, the primary purpose of Bobax is sending spam. With the help of Bobax, an automated spamming network can be setup very easily. A detailed analysis of Bobax was published by Stewart [Gro04a].

Other Interesting Bots. Presumably one of the most widespread bots is Toxbot. It is a variant of Codbot, a widespread family of bots. Common estimations of the botnet size achieved by Toxbot reach from a couple of hundred thousand compromised machines to more than one million. Giving an exact number is presumably not possible, but it seems like Toxbot was able to compromise a large number of hosts [Sco07].

Another special bot is called aIRCBot. It is very small – only 2560 bytes. It is not a typical bot because it only implements a rudimentary remote control mechanism. The bot only understands raw IRC commands. In addition, functions to spread further are completely missing. But due to its small size, it can nevertheless be used by attackers.

Q8Bot and kaiten are very small bots, consisting of only a few hundred lines of source code. Both have one additional noteworthiness: they are written for Unix/Linux systems. These programs implement all common features of a bot: dynamic updating via HTTP-downloads, various DDoS-attacks (e.g., SYN-flooding and UDP-flooding), a remote control mechanism, and many more. In the version we have analyzed, spreaders are missing, but presumably versions of these bots exist that also include mechanisms to propagate further.

There are many different version of very simple bots based on the programming language Perl. These bots contain, in most cases, only a few hundred lines of source code and offer only a rudimentary set of commands (most often only DDoS attack capabilities). This type of bots is often used in attacks against Unix-based systems with *remote file inclusion* (RFI) vulnerabilities, i.e., a specific class of attacks in which the attacker can include a remote file in a web application and thus compromise a system.

4.3.3 Overview of Botnets

General Characteristics. After having introduced different characteristics of bots, we now focus on how attackers use the individual bots to form botnets. Usually, the controller of the botnet, the so called *botmaster*, compromises a series of systems using various tools and then installs a bot to enable remote control of the victim computer. As communication protocol for this remote command channel, attackers commonly use

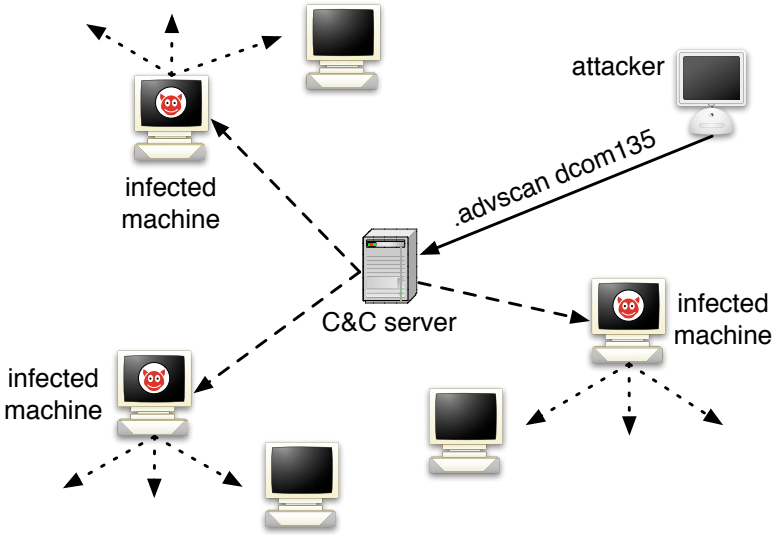


Figure 4.1: Typical setup for botnet with central server for Command & Control. The server can use IRC, HTTP, or a custom protocol.

IRC or HTTP, but also other — sometimes even proprietary — communication protocols can be used to send commands to the infected machines.

A typical setup of a botnet is shown in Figure 4.1. A central server is used for C&C: the attacker sends the command to the C&C server, which disseminates the command to the bots. For IRC-based botnets, the procedure works as follows: the bots connect to the IRC server at a predefined port and join a specific channel. The attacker can issue commands in this channel, and these commands are sent via the C&C server to the individual bots, which then execute these commands. In such a setup, the communication channel is thus a *push* mechanisms. In contrast, HTTP-based botnets use a *pull* mechanism: periodically, each bot requests a specific URL, which encodes status information, from the C&C server. This can, for example, look like the following request:

```
GET /cgi-bin/get.cgi?port=5239&ID=866592496&OS=WindowsXP&
                    conn=LAN&time=10:04:19
```

The infected machines tries to reach a web server running at a certain IP address. A CGI script is used as communication endpoint. Within the parameters of the script, the bot encodes several information like the port on which a backdoor is opened or information about the current connection type. As a reply, the server sends to the infected machine the command it should execute.

In the following, we first focus on IRC-based botnets and later on cover botnets that use HTTP for Command and Control.

Botnet Propagation. Most bots can automatically scan whole network ranges and propagate themselves using vulnerabilities and weak passwords on other machines.

After successful invasion, a bot uses TFTP, FTP, HTTP, CSend (a custom protocol used by some bots to send files to other users), or another custom protocol to transfer itself to the compromised host. The binary is started and tries to connect to the hard-coded master IRC server on a predefined port, often using a server password to protect the botnet infrastructure. This server acts as the C&C server to manage the botnet. Often a dynamic DNS name is provided rather than a hard-coded IP address, so the server can be relocated by the attacker. Using a specially crafted nickname, the bot tries to join the master's channel, often using a channel password, too. In this channel, the bot can be remotely controlled by the attacker.

Commands can be sent to the bot in two different ways: via sending an ordinary command directly to the bot or via setting a special topic in the command channel that all bots interpret. For example, the topic

```
.advscan dcom135 25 5 0 -c -s
```

tells the bots to spread further with the help of a known vulnerability (the Windows DCOM vulnerability [Mic03]) on TCP port 135. The bots start 25 concurrent threads that scan with a delay of 5 seconds for an unlimited amount of time (parameter 0). The scans target machines within the same Class C network of the bot (parameter `-c`) and the bots are silent (parameter `-s`), i.e., they do not send any report about their activity to the botmaster. As another example, the topic

```
.update http://<server>/BaxTer.exe 1
```

instructs the bots to download a binary from the Internet via HTTP to the local filesystem and execute it (parameter 1). Finally, as a third example, the command

```
.ddos.ack 85.131.xxx.xxx 22 500
```

orders the bots to attack a specific IP address with a DDoS attack. All bots send packets to the specified IP address on TCP port 22 for 500 seconds.

If the topic does not contain any instructions for the bot, then it does nothing but idle in the channel, awaiting commands. That is fundamental for most current bots: they do not spread if they are not told to spread in their master's channel.

To remotely control the bots, the controller of a botnet has to authenticate himself before issuing commands. This authentication is done with the help of a classical authentication scheme: at first, the controller has to log in with her username. Afterward, she has to authenticate with the correct password to approve her authenticity. The whole authentication process is usually only allowed from a predefined domain, thus only authorized people can start this process. Once an attacker is authenticated, she has complete control over the bots and can execute arbitrary commands.

4.4 Capturing Samples of Autonomous Spreading Malware

The main tool to collect malware in an automated fashion today are so-called *honeypots*. We have introduced honeypots in detail in Section 2.2 and for the sake of completeness

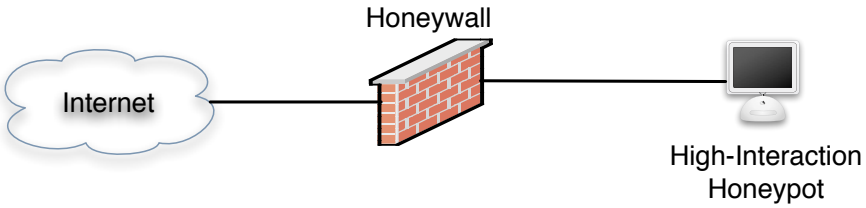


Figure 4.2: Honeypot setup for tracking botnets

we briefly summarize the main technical concepts of honeypots we use for botnet tracking. A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource [Mai03]. A *honeynet* is a network of honeypots. The idea behind this methodology is to lure in attackers such as automated malware and then study them in detail. The literature distinguishes two general types of honeypots: *High-interaction honeypots* offer the attacker a real system to interact with, while *low-interaction honeypots* only *emulate* network services or the network stack of an operating system.

Using a so called *GenIII Honeynet* [BV05] containing a high-interaction based Windows honeypot, we can already study botnets. We deploy a typical GenIII Honeynet with some small modifications as depicted in Figure 4.2. The high-interaction honeypot runs an unpatched version of Windows 2000 or Windows XP and is thus very vulnerable to attacks. It is located within the internal network of the university, but no filtering of the network traffic takes place. On average, the expected lifespan of the honeypot is less than ten minutes according to our empirical results. After this small amount of time, the honeypot is often successfully exploited by automated malware.

After a bot has exploited a vulnerability in our honeypot, it contacts the C&C server to obtain commands. Since we can observe the in- and outgoing network traffic to the honeypot, we can also detect and analyze this communication of the bot with the C&C server. Based on this analysis, we can derive information about the network location of the C&C server, the network port used for communication, and other information such as the protocol used. By analyzing the hard disk of the honeypot, we can also extract a sample of the bot, enabling a way to collect samples of autonomous spreading malware. An even more efficient approach to collect samples of autonomous spreading malware with high-interaction honeypots that uses the same technique as outlined above is HoneyBow [ZHH+07].

The approach described above works in practice, but has several drawbacks:

- A honeypot will crash regularly if the bot fails to exploit the offered service, e.g., due to a wrong offset within the exploit.

- The honeypot has to be closely monitored in order to detect system changes. Furthermore, these changes have to be analyzed carefully to detect malware.

- The approach does not scale well; observing a large number of IP addresses is difficult and resource-intensive.

To overcome these limitations, we employ the idea behind low-interaction honeypots to collect autonomous spreading malware in an automated way. The main idea behind this approach is that such malware scans the network for vulnerable system and we just need to *pretend* to be vulnerable. It is sufficient to *emulate* vulnerable network services to trick the malware into thinking that our honeypot is actually vulnerable. The infected machine then attacks the honeypot and we emulate the whole infection process, including analyzing the shellcode and downloading a copy of the malware. At the end, we have thus collected a sample of the malware and our honeypot is not infected since we merely emulated the infection process.

Instead of providing or emulating a complex network service, the honeypot only emulates the parts of a service that are vulnerable to remote attacks. This helps for scalability and reliability as we explain in the following paragraphs. Currently, there are two main concepts in this area: honeyd scripts simply emulate the necessary parts of a service to fool automated tools or very low-skilled attackers [Pro04]. This allows a large-scale deployment with thousands of low-interaction honeypots in parallel. But this approach has some limits: with honeyd it is not possible to emulate more complex protocols, e.g., a full emulation of FTP data channels is not possible. In contrast to this, high-interaction GenIII honeypots use a real system and thus do not have to emulate a service. The drawbacks of this approach have been described above. Deploying several thousand of these high-interaction honeypots is not possible due to limitations in maintenance and hardware requirements. Virtual approaches like Potemkin [VMC+05] look very promising, but are unfortunately not publicly available.

The gap between these approaches can be filled with the help of a special kind of low-interaction honeypots that emulate vulnerabilities in network services. This approach allows to deploy several thousands of honeypots in parallel with only moderate requirements in hardware and maintenance. With such a setup, we can collect information about malicious network traffic caused by autonomous spreading malware like bots in an efficient and scalable way.

Several tools implement the ideas presented above. *Nepenthes* was the first implementation for which a detailed description exists [BKH+06]. Göbel implemented *Amun*, which basically has the same scope and general designs as Nepenthes, with some enhancements in the area of emulating vulnerable network services [Göb08]. Within the scope of a diploma thesis, Trinius implemented a Windows version of the approach called *Omnivora* that enables collection of autonomous spreading malware on a machine running Microsoft Windows [Tri07]. In the following, we describe the design of such a low-interaction honeypot in more technical detail and use Nepenthes as the running example. The technical details for the two other implementations vary slightly, but the overall design is the same.

4.4.1 System Design of Nepenthes

Nepenthes is based upon a flexible and modularized design. The core — the actual daemon — handles the network interface and coordinates the actions of the other modules. The actual work is carried out by several modules, which register themselves in the Nepenthes core. Currently, there are several different types of modules:

- *Vulnerability modules* emulate the vulnerable parts of network services.
- *Shellcode parsing modules* analyze the payload received by one of the vulnerability modules. These modules analyze the received shellcode, an assembly language program, and extract information about the propagating malware from it.
- *Download modules* use the information extracted by the shellcode parsing modules to download the malware from a remote location.
- *Submission modules* take care of the downloaded malware, e.g., by saving the binary to a hard disk, storing it in a database, or sending it to antivirus vendors.
- *Logging modules* log information about the emulation process and help in getting an overview of patterns in the collected data.

In addition, several further components are important for the functionality and efficiency of the Nepenthes platform: *shell emulation*, a *virtual filesystem* for each emulated shell, *sniffing modules* to learn more about new activity on specified ports, and *asynchronous DNS resolution*.

Detailed Overview

The schematic interaction between the different components is depicted in Figure 4.3 and we introduce the different building blocks in the next paragraphs.

Vulnerability modules are the main building block of the Nepenthes platform. They enable an effective mechanism to collect malware. The main idea behind these modules is the following observation: in order to get infected by autonomous spreading malware, it is sufficient to only emulate the *necessary* parts of a vulnerable service. Thus instead of emulating the whole service, we only need to emulate the relevant parts and thus are able to efficiently implement this emulation. Moreover, this concepts leads to a scalable architecture and the possibility of large-scale deployment due to only moderate requirements on processing resources and memory. Often the emulation can be very simple: we just need to provide some minimal information at certain offsets in the network flow during the exploitation process. This is enough to fool the autonomous spreading malware and make it believe that it can actually exploit our honeypot. This is an example of the deception techniques used in honeypot-based research. With the help of vulnerability modules we trigger an incoming exploitation attempt and eventually we receive the actual payload, which is then passed to the next type of modules.

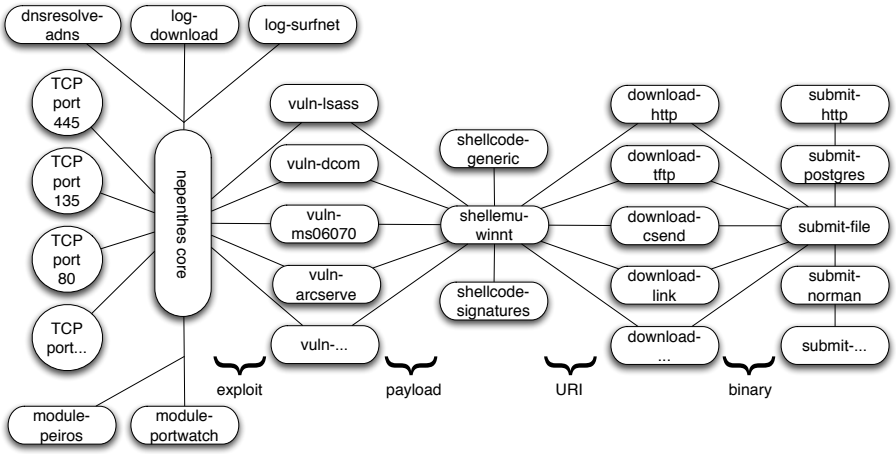


Figure 4.3: Schematic overview of Nepenthes platform

Shellcode parsing modules analyze the received payload and extract automatically relevant information about the exploitation attempt. These modules work in the following way: first, they try to decode the shellcode. Most of the shellcodes are encrypted with an *XOR encoder*. An XOR decoder is a common way to encrypt the actual shellcode in order to evade intrusion detection systems and avoid string processing functions. Afterwards the modules decode the code itself according to the computed key and then apply some pattern detection, e.g., `CreateProcess()` or generic URL detection patterns. The results are further analyzed (e.g., to extract credentials) and if enough information can be reconstructed to download the malware from the remote location, this information is passed to the next kind of modules.

Download modules have the task of downloading files from the remote location. Currently, there are seven different download modules. The protocols TFTP, HTTP, FTP and csend/creceive (a bot-specific submission method) are supported. Since some kinds of autonomous spreading malware use custom protocols for propagation, there are also download modules to handle these custom protocols. Fetching files from a remote location implies that the system running the low-interaction honeypot contacts other machines on the Internet. From an ethical point of view, this could be a problem since systems not under our control are contacted. A normal computer system that is infected by autonomous spreading malware would react in the same way, therefore we have no concerns downloading the malware from the remote location. However, it is possible to turn off the download modules. Then the system collects information about exploitation attempts and can still be useful as some kind of warning system.

In the last step, *submission modules* handle successfully downloaded files. Different modules allow a flexible handling of the downloaded binary, amongst others the following mechanisms are implemented:

- A module that stores the file in a configurable location on the filesystem.

- A module that submits the file to a central database to enable distributed sensors with central logging interface.
- A module that submits the file to a sandbox for automated analysis (see Section 4.5 for details).

Certain malware does not spread by downloading shellcodes, but by providing a shell to the attacker (so called *connect-back shellcode*). Therefore it is sometimes required to spawn and emulate a Windows shell such that the exploit is successful. Nepenthes offers *shell emulation* by emulating a rudimentary Windows shell to enable a shell interaction for the attacker. Several commands can be interpreted and batch file execution is supported. Such a limited simulation has proven to be sufficient to trick automated attacks. Based on the collected information from the shell session, it is then possible to also download the corresponding malware.

A common technique to infect a host via a shell is to write commands for downloading and executing malware into a temporary batch file and then execute it. Therefore, a *virtual filesystem* is implemented to enable this type of attacks. This helps in scalability since files are only created on demand, similar to the *copy-on-write* mechanism in operating systems: when the incoming attack tries to create a file, this file is created on demand and subsequently, the attacking process can modify and access it. All this is done virtually, to enable a higher efficiency. Every shell session has its own virtual filesystem, so that concurrent infection sessions using similar exploits do not interfere with each other. The temporary file is analyzed after the attacking process has finished and based on this information, the malware is downloaded from the Internet automatically. This mechanism is similar to *cages* in Symantec's ManTrap honeypot solution [Sym06].

Capturing New Exploits

An important factor of a honeypot-based system is also the ability to detect and respond to *zero-day attacks*, i.e., attacks that exploit an unknown vulnerability or at least a vulnerability for which no patch is available. The Nepenthes platform also has the capability to respond to this kind of threat. The two basic blocks for this ability are the *portwatch* and *bridging* modules. These modules can track network traffic at network ports and help in the analysis of new exploits. By capturing the traffic with the help of the portwatch module, we can at least learn more about any new threat, since we have already a full network capture of the first few packets. In addition, Nepenthes can be extended to really handle zero-day attacks. If a new exploit targets the Nepenthes platform, it will trigger the first steps of a vulnerability module. At some point, the new exploit will diverge from the emulation. This divergence can be detected, and then we perform a switch (hot swap) to either a real honeypot or some kind of specialized system for dynamic taint analysis (e.g., Argos [PSB06]). This second system is an example of the system for which Nepenthes is emulating vulnerabilities and with which it shares the internal state. This approach is similar to shadow honeypots [ASA+05].

An actual implementation of capturing novel attacks with Nepenthes was implemented by Leita as part of his PhD thesis [Lei08]. It is based on the idea of integrating Nepenthes

with *ScriptGen* and *Argos*. ScriptGen is a tool able to create simple emulators for any network protocol using a given network traffic sample of interaction. This means that ScriptGen observes the communication of two systems and can then generate an emulator of the protocol. No assumptions are made about the nature of the protocol, thus it can be applied to a very wide range of different protocols without any a priori knowledge of their protocol structure. More information about ScriptGen is available in a paper by Leita et al. [LMD05]. Argos is a high-interaction honeypot that uses the technique of *dynamic taint analysis* [NS05] to detect attacks that influence the control flow of a system. By focussing on the *effect* of an exploit and not the malicious code that is executed after the exploit was successful, Argos can detect unknown attacks without any signatures. The basic idea behind the whole system is to use finite state machines to handle known attacks and only use the other components for unknown attacks. This is achieved by coupling the capability of Argos to detect zero-day attacks with Nepenthes' capability to efficiently detect shellcode. Both systems are linked with the help of ScriptGen: this tool allows us to learn the behavior of a network protocol given some samples of real network communication provided by Argos. A similar system to handle zero-day attacks could presumably also be built with the honeypot solutions proposed by Cui [Cui06].

Limitations

We also identified several limitations of the low-interaction honeypots that emulate vulnerabilities in network services, which we present in this section. First, this kind of honeypots is only capable of collecting malware that is spreading autonomously — that is, that propagates further by scanning for vulnerable systems and then exploits them. We can thus not collect rootkits or Trojan horses with this tool, since these kinds of malware normally have no ability to propagate on their own. This is a limitation that this solution has in common with most other honeypot-based approaches. A website that contains a browser exploit that is only triggered when the website is accessed will not be detected with ordinary honeypots due to their passive nature. The way out of this dilemma is to use *client-side honeypots* like HoneyMonkeys [WBJ+06], HoneyClient [Wan09], or Capture-HPC [SS09] to detect these kinds of attacks. The modular architecture of Nepenthes would enable this kind of vulnerability modules, but this is not the aim of the Nepenthes platform. The empirical results show that Nepenthes is able to collect many different types of bots, even without this kind of modules.

Malware that propagates by using a hitlist to find vulnerable systems that can be attacked [SMPW04] is hard to detect with low-interaction honeypots as presented in this section. This is a limitation that Nepenthes, Amun and Omnivora have in common with all current honeypot-based systems and also other approaches in the area of vulnerability assessment. Here, the solution to the problem would be to become part of the hitlist. If, for example, the malware generates its hitlist by querying a search engine for vulnerable systems, the trick would be to smuggle a honeypot system in the index of the search engine. Currently, it is unclear how such an advertisement could be implemented within the low-interaction honeypots. But there are other types of honeypots that can be used

to detect hitlist-based malware, such as honeypot solution that use search engines to become part of the hitlist (e.g., Google Hack Honeypot [Rya08] or HIHAT [Müt07]) .

It is possible to remotely detect the presence of the low-interaction honeypots: since such a honeypot instance normally emulates a large number of vulnerabilities and thus opens many TCP ports, an attacker could become suspicious during the reconnaissance phase. Current automated malware does not check the plausibility of the target, but future malware could do so. To mitigate this problem, the stealthiness can be improved by using only the vulnerability modules that belong to a certain configuration of a real system, for example, only vulnerability modules that emulate vulnerabilities for Windows 2000 Service Pack 1. The tradeoff lies in reduced expressiveness and leads to fewer samples collected. A similar problem with stealthiness appears if the results obtained by running such a honeypot are published unmodified. To mitigate such a risk, we refer to the solution outlined in a paper by Shinoda et al. [SII05].

4.4.2 Evaluation

Vulnerability Modules

Vulnerability modules are one of the most important components of the whole honeypot approach, since they take care of the emulation process. For Nepenthes, there are more than 20 vulnerability modules in total. They cover well-known, but old vulnerabilities such as the one related to buffer overflows in the Windows RPC interface (DCOM vulnerability [Mic03] or Lsasrv.dll RPC buffer overflow [Mic04]). Furthermore, also recent vulnerabilities such as a vulnerability in the Server service (MS08-067 [Mic08]) are emulated. Besides these vulnerabilities in the Windows OS, also vulnerabilities in third-party applications are emulated by Nepenthes. Other honeypot solutions with the same goal such as Amun or Omnivora also emulate different kinds of vulnerabilities and the expressiveness of the different solutions is comparable.

This selection of emulated vulnerabilities has proven to be sufficient to handle most of the autonomous spreading malware we have observed in the wild. As we show in the remainder of this chapter, these modules allows us to learn more about the propagating malware. However, if a certain packet flow cannot be handled by any vulnerability module, all collected information is stored on hard disk to facilitate later analysis. This allows us to detect changes in attack patterns, highlights new trends, and helps us develop new modules. In the case of a zero day attack this can enable a fast analysis because the first stages of the attack have already been captured.

Scalability

In this section, we evaluate the scalability of the Nepenthes platform. With the help of several metrics, we determine how effective this approach is and how many honeypot systems can be emulated on a single physical machine. We focus on Nepenthes since this honeypot solution scales best. Both Amun and Omnivora do not scale as good as Nepenthes, mainly due to the following reasons. First, Amun is implemented in Python,

a scripting language. As a result, the implementation is slower, but it can nevertheless reach about 50% of the scalability of Nepenthes. Second, Omnivora is implemented for Windows systems and thus has some intrinsic limitations regarding the scalability of the implementation [Tri07].

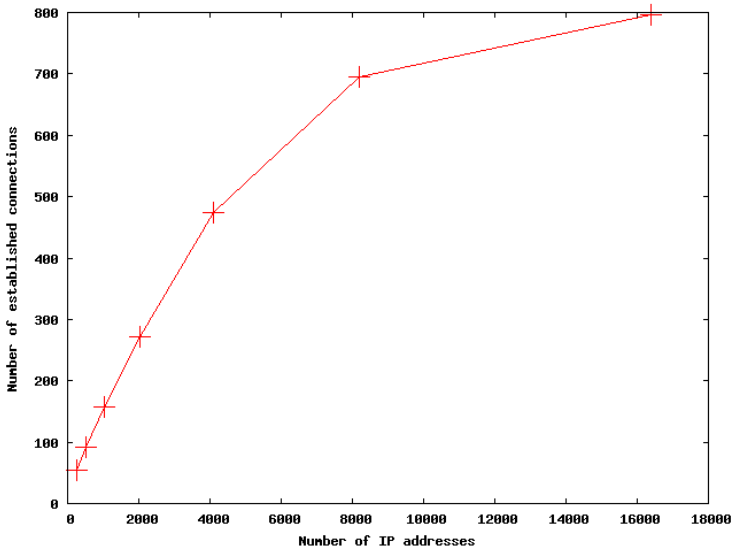
As noted in the paper about Potemkin [VMC+05], a "key factor to determine the scalability of a honeypot is the number of honeypots required to handle the traffic from a particular IP address range". To cover a /16 network, a naive approach would be to install over 64,000 ordinary honeypots to cover the whole network range. This would, of course, be a waste of resources, since only a limited number of IP addresses receive network traffic at any given point in time. Honeyd can simulate a whole /16 network on just a single computer [Pro04] and Nepenthes scales comparably well.

To evaluate the scalability of Nepenthes, we have used the following setup. The testbed is a commercial off-the-shelf (COTS) system with a 2.4GHz Pentium III, 2 GB of physical memory, and 100 MB Ethernet NIC running Debian Linux 3.0 and version 2.6.12 of the Linux kernel. This system runs Nepenthes 0.2 in default configuration. This means that all 21 vulnerability modules are used, resulting in a total of 29 TCP sockets on which Nepenthes emulates vulnerable services.

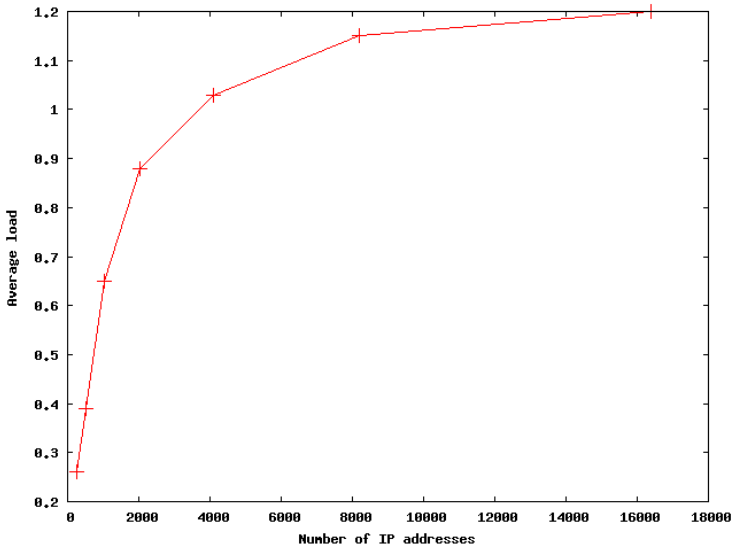
We tested the implementation with different quantities of emulated systems, ranging from only 256 honeypots up to 32,000 emulated honeypots. For each configuration, we measured the number of established TCP connections, the system load, and the memory consumption of Nepenthes for one hour. We repeated this measurement several times in different order to cancel out statistical unsteadiness. Such an unsteadiness could, for example, be caused by diurnal properties of malware epidemics [DZL06] or bursts in the network traffic. The average value of all measurements is then an estimation of the specific metric we are interested in. Figure 4.4 provides an overview of our results. In both graphs, the x-axis represents the number of IP addresses assigned to Nepenthes running on the testbed machine. The y-axis represents (a) the number of established TCP connections and (b) the average system load, respectively. We did not plot the memory consumption because it is so low (less than 20 MB for even a large number of simulated IP addresses) and nearly independent from the number of established TCP connections. In (a) we see that the scalability is nearly linear up to 8192 IP addresses. This corresponds to the system load, which is below 1 (b). Afterward, the number of established TCP connections is decreasing, which is caused by a system load above 1 — that is, the system is fully occupied with I/O operations.

4.4.3 Related Work

Large-scale measurements of malicious network traffic have been the focus of previous research. With the help of approaches like the *network telescope* [MSVS04] or *darknets* [Cym06] it is possible to observe large parts of the Internet and monitor malicious activities. In contrast to Nepenthes, these approaches *passively* collect information about the network status and can infer further information from it, e.g., inferring the amount of Distributed Denial-of-Service attacks [MVS01]. By not responding to the packets, it is not possible to learn more about full attacks. Slightly more expressive approaches



(a) Number of concurrent established TCP connections



(b) System load

Figure 4.4: Measurement results for scalability of Nepenthes in relation to number of IP addresses assigned to the sensor.

like the *Internet Motion Sensor* [BCJ+05] differentiate services by replying to a TCP SYN packet with TCP SYN-ACK packets. However, their expressiveness is also limited and only with further extensions it is possible to also learn more about spreading malware.

Honeyd [Pro04] creates virtual hosts on a network. It simulates the TCP/IP stack of arbitrary operating systems and can be configured to run arbitrary services. These services are generally small scripts that emulate real services, and offer only a limited

expressiveness. Honeyd can simulate arbitrary network topologies including dedicated routes and routers, and can be configured to feign latency and packet loss. In contrast to Nepenthes, honeyd does not offer as much expressiveness since the reply capabilities of honeyd are limited from a network point of view. Nepenthes can be used as a subsystem for honeyd, however. This extends honeyd and enables a way to combine both approaches: Nepenthes acts then as a component of honeyd and is capable of dealing with automated downloading of malware.

The *Collapsar platform* [JX04] is a virtual-machine-based architecture for network attack detention. It allows to host and manage several high-interaction virtual honeypots in a central network. Malicious traffic is redirected from other networks (*decentralized honeypot presence*) to this central network which hosts all honeypots (*centralized honeypot management*). This enables a way to build a *honeyfarm*. Note that the idea of a honeyfarm is not tied to the notion of a high-interaction honeypot: it is also possible to deploy Nepenthes or other low-interaction honeypots as a honeyfarm system by redirecting traffic from remote locations to a central Nepenthes server. In fact, the design of SGNET uses some of these ideas [LD08].

Internet Sink (*iSink*) [YBP04] is a system that passively monitors network traffic and is also able to actively respond to incoming connection requests. The design is stateless and therefore the expressiveness of the responses is limited. Similarly, *HoneyTank* [VBBC04] is a system that implements stateless responders to network probes. This allows to collect information about malicious activities to a limited amount. In contrast to these systems, Nepenthes implements a finite state machine to emulate vulnerabilities. This allows us to collect more detailed information about an attack.

Potemkin [VMC+05] exploits virtual machines, aggressive memory sharing, and late binding of resources to emulate more than 64,000 high-interaction honeypots using ten physical servers. This approach is promising, but has currently several drawbacks compared to low-interaction honeypots: first, each honeypot within Potemkin has to be a fixed system in a fixed configuration. In contrast to this, the vulnerability modules of Nepenthes allow greater flexibility. As mentioned above, Nepenthes can react for example on exploitation attempts against Windows 2000 and Windows XP, even regardless of service pack. It would even be possible to emulate vulnerabilities for different operating systems and even different processor architectures on a single Nepenthes honeypot. Second, there are currently only preliminary results for the scalability of Potemkin. Vrable et al. provide only results for a representative 10 minutes period [VMC+05] and no implementation is publicly available.

4.5 Automated Malware Analysis

With the help of low-interaction honeypots that emulate vulnerabilities in network services we can collect a binary copy of autonomous spreading malware without any human interaction, thus this step is automated to a high degree. In order to learn more about the remote control structure behind such malware, we also need to automatically

analyze a collected binary: we need to extract all important information about a botnet and then we can start the actual infiltration process.

A possible way to extract the information from the captured malware is *reverse engineering*, the process of carefully analyzing a program without having its source code. This process is time consuming and error prone. A better approach is an automated analysis with the help of a honeynet. A standard honeynet setup can also be used for this purpose: upon startup, a Windows honeypot downloads a piece of malware from a database located within our university network. It executes the file and reboots itself after a few minutes. During this time span, the bot installs itself on the honeypots and connects to the C&C server. With the help of the Honeywall, we are again able to extract all necessary information like the hostname of the C&C server or the channel the bot joins in a fully automated way. During each reboot phase, the honeypot resets the hard disk so that a clean image is booted each time. Such a setup allows us to extract interesting information related to the botnet from a given binary, but since we want to analyze the sample in detail, a more advances mechanisms is necessary for automated malware analysis which we introduce in this section.

In our view the most efficient approach for an automated analysis of a collected binary in terms of (1) *automation*, (2) *effectiveness* and (3) *correctness* is a *sandbox*. Automation means that the analysis tool should create a detailed analysis report of a malware sample quickly and without user intervention. A machine readable report can in turn be used to initiate automated response procedures like updating signatures in an intrusion detection system, thus protecting networks from new malware samples on the fly. Effectiveness of a tool means that all relevant behavior of the malware should be logged, no executed functionality of the malware should be overlooked. This is important to realistically assess the threat posed by the malware sample. Finally, a tool should produce a correct analysis of the malware, i.e., every logged action should in fact have been initiated by the malware sample to avoid false claims about it.

As part of a diploma thesis, Willems developed a tool for automated malware analysis called *CWSandbox* [Wil06, WHF07] that we use for our work. CWSandbox fulfills the three design criteria of automation, effectiveness and correctness for the Windows family of operating systems:

- Automation is achieved by performing a *dynamic analysis* of the malware. This means that malware is analyzed by executing it within a controlled environment, which works for any type of malware in almost all circumstances. A drawback of dynamic analysis is that it commonly only analyses *a single* execution of the malware. Moser et al. introduced a way to explore multiple execution paths with dynamic analysis [MKK07], which circumvents this limitation.

  In contrast to dynamic analysis, the method of *static analysis* in which the source code is analyzed without actually executing it allows to observe *all* executions of the malware at once. Static analysis of malware, however, is rather difficult since the source code is commonly not available. Even if the source code were available, one could never be sure that no modifications of the binary executable happened, which were not documented by the source. Static analysis at the

machine code level is often extremely cumbersome since malware often uses code-obfuscation techniques like compression, encryption, or self-modification to evade decompilation and analysis. Since the aim of our research lies on automated analysis of malware binaries, we chose dynamic analysis.

- Effectiveness is achieved by using the technique of *API hooking*. API hooking means that calls to the Win32 application programmers' interface (API) are re-routed to the monitoring software before the actual API code is called, thereby creating insight into the sequence of system operations performed by the malware sample. API hooking ensures that all those aspects of the malware behavior are monitored for which the API calls are hooked. API hooking therefore guarantees that system level behavior (which at some point in time is commonly an API call) is not overlooked unless the corresponding API call is not hooked.

  API hooking can be bypassed by programs which directly call kernel code in order to avoid using the Windows API. However, this is rather uncommon in malware, as the malware author needs to know the target operating system, its service pack level, and some other information in advance. Our empirical results show that most autonomous spreading malware is designed to attack a large user base and thus commonly uses the Windows API.

- Correctness of the tool is achieved through the technique of *DLL code injection*. Roughly speaking, DLL code injection allows API hooking to be implemented in a modular and reusable way, thereby raising confidence in the implementation and the correctness of the reported analysis results.

The combination of these three techniques within CWSandbox allows to trace and monitor all relevant system calls and generate an automated, machine-readable report that describes for example

- which files the malware sample has created or modified,
- which changes the malware sample performed on the Windows registry,
- which dynamic link libraries (DLLs) were loaded before executing,
- which virtual memory areas were accessed,
- which processes were created, and
- which network connections were opened and what information was sent over such connections.

Obviously, the reporting features of the CWSandbox cannot be perfect, i.e., they can only report on the visible behavior of the malware and not on how the malware is programmed. Using CWSandbox also entails some danger which arises from *executing* malicious software on a machine which is connected to a network. However, the information derived from executing malware for even very short periods of time in the CWSandbox environment is surprisingly rich and in most cases sufficient to assess the danger originating from the malware sample.

4.5.1 Technical Background

To understand the system design of CWSandbox, we provide in this section an overview of the four building blocks of CWSandbox, namely *dynamic malware analysis, API hooking*, *inline code overwriting*, and *code injection*.

Dynamic Malware Analysis

Dynamic analysis means to observe one or more behaviors of a software artifact to analyze its properties by executing the software itself. We have already argued above that dynamic analysis is preferable to static (code) analysis when it comes to malware. There exist two different approaches to dynamic malware analysis with different result granularity and quality:

1. taking an image of the complete system state before and comparing this to the complete system state right after the malware execution

2. monitoring all actions of the malware application during its execution, e.g., with the help of a debugger or a specialized tool

It is evident that the first option is easier to implement, but delivers more coarse-grained results, which sometimes are sufficient, though. This approach can only analyze the cumulative effects and does not take dynamic changes into account. If for example a file is generated during the malware's execution and this file is deleted before the malware terminates, the first approach will not be able to observe this behavior. The second approach is harder to implement, but delivers much more detailed results, so we chose to use this approach within CWSandbox.

API Hooking

The *Windows API* is a programmer's interface which can be used to access the Windows resources, e.g., files, processes, network, registry, and all other major parts of Windows. User applications use the API instead of making direct system calls and thus this offers a possibility for behavior analysis: we obtain a dynamic analysis if we monitor all relevant API calls and their parameters. The API itself consists of several DLL files that are contained in the Windows System Directory. Some of the most important files are `kernel32.dll`, `advapi32.dll`, `ws2_32.dll`, and `user32.dll`. Nearly all API functions do not call the system directly, but are only wrappers to the so called *Native API* which is implemented in the file `ntdll.dll`. With the Native API, Microsoft introduces an additional API layer. By that, Microsoft increases the portability of Windows applications: the implementation of native API functions can change from one Windows version to another, but the implementation and the interface of the regular Windows API functions are more stable over time.

The Native API is not the end of the execution chain which is performed when an API function is executed. Like in other operating systems, the running process has to switch

from *usermode* (Ring 3) to *kernelmode* (Ring 0) in order to perform operations on the system resources. This is mostly done in the `ntdll.dll`, although some Windows API functions switch to kernelmode by themselves. The transfer to kernelmode is performed by initiating a software interrupt, Windows uses `int 0x2e` for that purpose, or by using processor specific commands, i.e., `sysenter` for Intel processors or `syscall` for AMD processors. Control is then transfered to `ntoskrnl.exe` which is the core of the Windows operating system.

In order to observe the control flow of a given malware sample, we need to somehow get access to these different API function. A possible way to achieve this is *hooking*. Hooking of a function means the interception of any call to it. When a hooked function should be executed, control is delegated to a different location, where customized code resides: the *hook* or *hook function*. The hook can then perform its own operations and later transfer control back to the original API function or prevent its execution completely. If hooking is done properly, it is hard for the calling application to detect that the API function was hooked and that the hook function was called instead of the original one. However, the malware application could try to detect the hooking function and thus we need to carefully implement it and try to hide as good as possible the analysis environment from the malware process.

After the execution of the hook function, control is delegated back to the caller, so that the hook is transparent to it. Analog to that, *API Hooking* means the interception of API calls. This techniques allows the hook function to be informed any time a given API function is called. Additionally, it can analyze the calling parameters used and modify them if necessary. Then it can call the original API function with altered parameters, modify the results of it, or even suppress the call of the original API function at all.

There are several methods that allow the interception of system calls during their way from a potentially malicious user application to the ultimate kernel code [Iva02]. One can intercept the execution chain either inside the user process itself, in one or multiple parts of the Windows API or inside the Windows kernel by modifying the *Interrupt Descriptor Table* (IDT) or the *System Service Dispatch Table* (SSDT). All of them have different advantages, disadvantages, and complexity. We use the technique of *Inline Code Overwriting* since it is one of the most effective and efficient methods.

Inline Code Overwriting

With Inline Code Overwriting, the code of the API functions, which is contained in the DLLs loaded into the process memory, is overwritten directly. Therefore, *all* calls to these APIs are re-routed to the hook function, no matter at what time they occur or if those are linked implicitly or explicitly. We perform the inline code overwriting with the following five steps:

1. The target application is created in suspended mode. This means that the Windows loader loads and initializes the application and all implicitly linked DLLs, but does not start the main thread, such that no single operation of the application is performed.

2. When all the initialization work is done, every to be hooked function is looked up in the Export Address Table (EAT) of the containing DLL and their code entry points are retrieved.

3. As the original code of each hooked API function will be overwritten, we have to save the overwritten bytes in advance, as we later want to reconstruct the original API function.

4. The first few instructions of each API function are overwritten with a JMP (or a CALL) instruction leading to the hook function.

5. To make this method complete, the API functions that allow the explicit binding of DLLs (`LoadLibrary` and `LoadLibraryEx`) also need to be hooked. If a DLL is loaded dynamically at runtime, the same procedure as above is performed to overwrite the function entry points, before control is delegated back to the calling application.

Of course the hook function does not need to call the original API function. Also there is no need to call it with a JMP: the hook function can call the original API with a CALL operation and get back control when the RET is performed in the called API function. The hook function can then analyze the result and modify it, if this is necessary.

One of the most popular and detailed descriptions of this approach is available in an article published in the Code Breakers Journal [Fat04]. Microsoft also offers a library for that purpose, called *Detours* [HB99].

For completeness reasons, we also mention *System Service Hooking*. This technique performs hooking at a lower level within the Windows operating system and is thus not considered as API Hooking. There are two additional possibilities for rerouting API calls. On the one hand, an entry in the *Interrupt Descriptor Table* (IDT) can be modified, such that interrupt `int 0x2e`, which performs the transition from usermode to kernelmode, points to the hooking routine. On the other hand, the entries in the *System Service Dispatch Table* (SSDT) can be manipulated, such that the system calls can be intercepted depending on the service IDs. We do not use these techniques for now, since API hooking has proven to deliver accurate results in practice. However, in the future we may extend CWSandbox to also use kernel hooks since this is more complicated to detect and offers an additional point of view to monitor the execution of a given sample.

Code Injection

API hooking with inline code overwriting makes it necessary to patch the application after it has been loaded into memory. To be successful, we perform the following two steps:

- We copy the hook functions into the target application's address space, such that these can be called from within the target; this is the actual code injection.

- We also have to bootstrap and set up the API hooks in the target application's address space using a specialized thread in the malware's memory.

For installing the hooks, the performed actions depend on the hooking method used. In any case, the memory of the target process has to be manipulated, e.g., by changing the *import address table* (IAT) of the application itself, changing the *export address table* (EAT) of the loaded DLLs, or directly overwriting the API function code. Windows offers functions to perform both of the necessary tasks for implanting and installing API hook functions: accessing another process' virtual memory and executing code in a different process' context.

Accessing the virtual memory of another process is possible: `kernel32.dll` offers the API functions `ReadProcessMemory` and `WriteProcessMemory`, which allow the reading and writing of an arbitrary process' virtual memory. Of course, the reader and writer need appropriate security privileges, respectively. If he holds them, he even can allocate new memory or change the protection of an already allocated memory region by using `VirtualAllocEx` and `VirtualProtectEx`.

Executing code in another process' context is possible in at least two ways:

1. suspend one running thread of the target application, copy the code to be executed into the target's address space, set the instruction pointer of the resumed thread to the location of the copied code, and then resume this thread, or

2. copy the code to be executed into the target's address space and then create a new thread in the target process with the code location as the start address.

Both techniques can be implemented with appropriate API functions. With those building blocks it is now possible to inject code into another process.

The most popular technique for code injection is the so called *DLL injection*. All custom code is put into a DLL, called the *injected DLL*, and the target process is directed to load this DLL into its memory space. Thus, both requirements for API hooking are fulfilled: the custom hook functions are loaded into the target's address space, and the API hooks can be installed in the DLL's initialization routine, which is called automatically by the Windows loader.

The explicit linking of a DLL is performed by the API functions `LoadLibrary` or `LoadLibraryEx`, from which the latter one simply allows some more options. The signature of the first function is very simple, the only parameter needed is a pointer to the name of the DLL. The trick is to create a new thread in the target's process context using the API function `CreateRemoteThread` and then setting the code address of the API function `LoadLibrary` as the starting address of this newly created thread: when the new thread is executed, the function `LoadLibrary` is called automatically inside the target's context. Since we know the location of `kernel32.dll` (always loaded at the same memory address) from our starter application and also know the code location of the `LoadLibrary` function, we can use these values for the target application.

4.5.2 System Design of CWSandbox

With the building blocks described in the previous section, we can now describe a system that is capable of automatically analyzing a given malware sample: *CWSandbox*. This

system performs a behavior-based analysis, i.e., the malware binary is executed in a controlled environment and all relevant function calls to the Windows API are observed. In a second step, a high level summarized report is generated from the monitored API calls. The analysis report contains a separate section for each process that was involved. For each process, there exist several subsections that contain associated actions, i.e., there is one subsection for all accesses to the filesystem and another section for all network operations. In the following, we describe the tool in detail.

The sandbox routes nearly all API calls to the original API functions, after it has analyzed their call parameters. Therefore, the malware is not blocked from integrating itself into the target operating system, e.g., by copying itself to the Windows system directory or adding new registry keys. To enable a fast automated analysis, we thus execute the CWSandbox in a special environment, so that after the completion of an analysis process the system can easily be brought back into a clean state. This is implemented by executing CWSandbox in a native environment, i.e., a normal commercial off-the-shelf system, and an automated procedure to restore a clean state.

Architecture

CWSandbox itself consists of two separate applications: *cwsandbox.exe* and *cwmonitor.dll*. The sandbox application creates a suspended process of the malware application and injects the DLL into it (*DLL injection*). At the initialization of this DLL, API hooks for all interesting API functions are installed (*API hooking*). The sandbox application then sends some runtime options to the DLL and the DLL in turn answers with some runtime information of the malware process. After this initialization phase, the malware process is resumed and executed for a given amount of time. During the malware's execution, all hooked API calls are re-routed to the referring hook functions in the DLL. These hook functions inspect the call parameters, inform the sandbox about the API call in form of a notification object, and then – depending on the type of the API function called – delegate control to the original function or return directly. If the original API is called, the hook function inspects the result and sometimes modifies it, before returning to the calling malware application. This is for example done to hide the presence of the sandbox: certain files, processes and registry keys which belong to the implementation of the sandbox are filtered out from the results and thus their existence is hidden from the sample under observation.

Besides the monitoring, the DLL also has to ensure that whenever the malware starts a new process or injects code into a running process, the sandbox is informed about this. The sandbox then injects a new instance of the DLL into that newly created or already existing process, so that all API calls from this process are also captured.

Inter-process Communication between Sandbox and the DLL

There is a lot of communication between the executable and all the loaded instances of the monitoring DLL. Since the communication endpoints reside in different processes, this communication is called *inter-process communication* (IPC). Each API hook func-

tion sends a notification object to inform the sandbox about the call and the calling parameters used. Some hook functions also require an answer from the sandbox which determines the further procedure, e.g., if the original API function should be called or not. A lot of data has to be transmitted per notification and a various number of instances of the DLL can exist, so there is a heavy communication throughput. Besides the high performance need, also a very reliable mechanism is needed, as no data is allowed to be lost or modified on its way. Thus, a reliable IPC mechanism with high throughput had to be implemented.

Implementation of cwsandbox.exe and cwmonitor.dll

The work of the sandbox can be divided into three phases:

1. initialization phase,
2. execution phase, and
3. analysis phase.

In the first phase, the sandbox initializes and sets up the malware process. It then injects the DLL and exchanges some initial information and settings. If everything worked well, the process of the malware is resumed and the second phase is started. Otherwise the sandbox kills the newly created malware process and also terminates. The second phase lasts as long as the malware executes, but can be ended prematurely by the sandbox. This happens if a timeout occurs or some critical conditions require an instant termination of the malware. During this phase, there is a heavy communication between the `cwmonitor.dll` instances in all running processes and `cwsandbox.exe`. In the third phase, all the collected data is analyzed and an XML analysis report is generated based on the collected information.

The `cwmonitor.dll` is injected by the sandbox into each process that is created or injected by the malware. The main tasks of the DLL are the installation of the API hooks, realization of the hook functions, and the communication with the sandbox. Similar to the sandbox, also the life cycle of the DLL can be divided into three parts *initialization*, *execution*, and *finishing* phase. The first and the last of these phases are handled in the DLL main function, the execution phase is handled in the different hook functions. Operations of the DLL are executed only during initialization and finishing phase and each time one of the hooked API functions is called.

Rootkit functionality

Since the malware sample should not be aware of the fact that it is executed inside of a controlled environment, CWSandbox implements some rootkit functionality: all system objects that belong to the sandbox implementation are hidden from the malware binary. In detail, these are processes, modules, files, registry entries, mutexes events, and handles in general. This at least makes it for the malware sample more difficult to detect the presence of the sandbox. Furthermore, additional components of the analysis

environment, e.g., the mechanisms to restore the infected system back to a clean state, are hidden as well. Up to now, we ran only in some cases into trouble with this approach, e.g., with samples that detect API hooking.

CWSandbox Summary

CWSandbox allows us to analyze a given binary without any human interaction. When the bot contacts the IRC server used for command and control, we can also observe all information related to the remote control infrastructure like the domain name of the C&C server or the channel the bot joined. CWSandbox can handle IRC communication on arbitrary network ports and is able to extract all the information we need in order to track an IRC-based botnet. Furthermore, also all information observed from HTTP communication is collected such that infiltration of HTTP-based botnets is feasible based on the dynamic analysis report. More information about CWSandbox is available in the thesis by Willems [Wil06] and a paper by Willems et al. [WHF07].

4.5.3 Related Work

Several tools for automatic behavior analysis of malicious software already exist. We briefly describe the *Norman Sandbox* and *TTAnalyze/Anubis* in this section and outline the difference between these tools and CWSandbox.

The *Norman SandBox* [Nor03] was developed by Norman ASA, a Norwegian company which has specialized in data security. In contrast to our solution, Norman *emulates* a whole computer and a connected network. This is archived by re–implementing the core Windows system and then executing the malware binary within this emulated environment. Implementation details, a description of the underlying technology, and a live demo can be found in a technical report [Nor09]. Compared to CWSandbox, the emulation has the advantage that this is transparent for the malware binary, i.e., the malware binary has no possibility to detect that it is executed within an emulated environment. But such an emulation has one limitation: the malware process cannot interfere with other running processes and infect or modify them, since there are no other processes within the emulation. However, our research shows that a significant amount of malware uses such techniques, e.g., by creating a remote thread within Internet Explorer and using it to download additional content from the Internet. By using a real operating system as the base of CWSandbox, the malware can infer with the system with only a very limited disturbance created via our API hooking.

Another comparable approach is *TTAnalyze/Anubis* [BKK06]. The major difference to our solution is that the technique of *virtual machine introspection* is used to observe the behavior of the to be analyzed malware sample. This had the main advantages that also techniques like multiple execution path analysis can be implemented [MKK07], which facilitates a more powerful analysis environment. Furthermore, Anubis is presumably more stealth since it uses the PC emulator QEMU [Bel05], which enables a tighter control over the malware sample. Our empirical results show that for most malware

samples, however, the analysis reports generated by CWSandbox and Anubis contain the same amount of information.

A different approach is Tomlin's *Sandnet*. The malicious software is executed on a real Windows system, not on an emulated or simulated one. After 60 seconds of execution, the host is rebooted and forced to boot from a Linux image. After booting Linux, the Windows partition is mounted and the Windows registry as well as the complete file list are extracted and the Windows partition is reverted back to its initial clean state. Since Sandnet focuses on network activity, several dispositions are made. During the execution of the malware, the Windows host is connected to a virtual Internet with an IRC server running which positively answers to all incoming IRC connection request. Furthermore, all packets are captured to examine all other network traffic afterwards. Compared to CWSandbox, the advantage of this approach is that a native operating system is used. But since only a snapshot of the infected system is taken, all dynamic actions, e.g., creation of new processes, cannot be monitored. *Truman* (*The Reusable Unknown Malware Analysis Net*) [Ste] is a similar approach.

Hunt et al. introduced Detours, a library for instrumenting arbitrary Windows functions [HB99]. With the help of this library it is possible to implement an automated approach for malware analysis similar to CWSandbox. We opted to implement our own API hooking mechanism in order to have greater flexibility and a more fine grained control over the instrumented functions.

4.6 Automated Botnet Infiltration

Once we have collected all sensitive information of the botnet, we start to infiltrate the botnet as we have all the necessary data to join a given botnet. In a first approach, it is possible to set up a normal IRC client and connect to the network. If the operators of the botnets do not detect this client, logging all the commands can be enabled. This way, all bot commands and all actions can be observed. If the botnet is relatively small, there is a chance that the bogus client will be identified, since it does not answer to valid commands. In this case, the operators of the botnets tend to either ban or attack the suspicious client using DDoS. But often it is possible to observe a botnet simply with a normal IRC client, to which we need to feed all information related to the botnet.

However, there are some problems with this approach. Some botnets use a very strongly stripped-down C&C server that is not standard compliant so that a normal IRC client cannot connect to this network. Furthermore, this approach does not scale very well. Tracking more than just a few botnets is not possible, since a normal IRC client will be overwhelmed with the amount of logging data, and it does not offer a concise overview of what is happening in all botnets.

Therefore, we use an IRC client optimized for botnet tracking called *botspy*. This software was developed by Overbeck in his diploma thesis [Ove07] and offers several techniques for observing botnets. It is inspired by the tool *drone*, developed by some members of the German Honeynet Project, and shares many characteristics with it:

- Multiserver support to track a large number of botnets in parallel

- Support for SOCKS proxies to be able to conceal the IP we are running the botnet monitoring software
- Database support to log all information collected by several botspy nodes in a central database
- Automated downloading of malware identified within the botnet
- Modular design to be flexible, e.g., to also support non-IRC-based or HTTP-based botnets

This tool allows us to monitor many botnets in parallel and infiltrate the botnets we found based on the automated analysis of CWSandbox. Rajab et al. [RZMT06] implemented an advanced version of botspy/drone which also emulates certain aspects of bot behavior. Such a more expressive approach enables a stealthier evasion of botnets since the botmaster cannot easily identify the presence of the agent within the botnet.

4.7 Botnet Tracking

With the building blocks described in the previous three sections, we can implement the general methodology presented in Chapter 3.

4.7.1 Tracking of IRC-based Botnets

To track classical botnets that use a central, IRC-based C&C server, we can use the three tools introduced above in a straightforward way: in the first phase, we capture samples of autonomous spreading malware with the help of honeypots, in our case either Nepenthes, HoneyBow, Amun, Omnivora, or any of the other honeypot solutions built for that use case. In the second phase, we use CWSandbox to generate a malware analysis report of a given sample. As a result, we obtain a dynamic analysis report that contains a summary of the behavior observed during runtime. This report commonly contains more information about the communication channel used by the bot, i.e., the domain name of the C&C server, the port to connect to, and the channel and nickname used by the bot. Based on this report, we can infiltrate the botnet and impersonate as a legitimate member of the botnet. Once we have joined the IRC channel used by the botnet, we can start observing all communication activity taking place in the channel. This enables us to observe all commands issued by the botmaster and we can — if the IRC server is configured to show bots entering and leaving the channel — also monitor how many victims belong to a given botnet. In the next section, we present empirical measurement results for tracking this kind of botnets.

4.7.2 Tracking of HTTP-based Botnets

As a second example, we show how our general methodology outlined in Chapter 3 can be used to also track other kinds of remote control networks. We now focus on

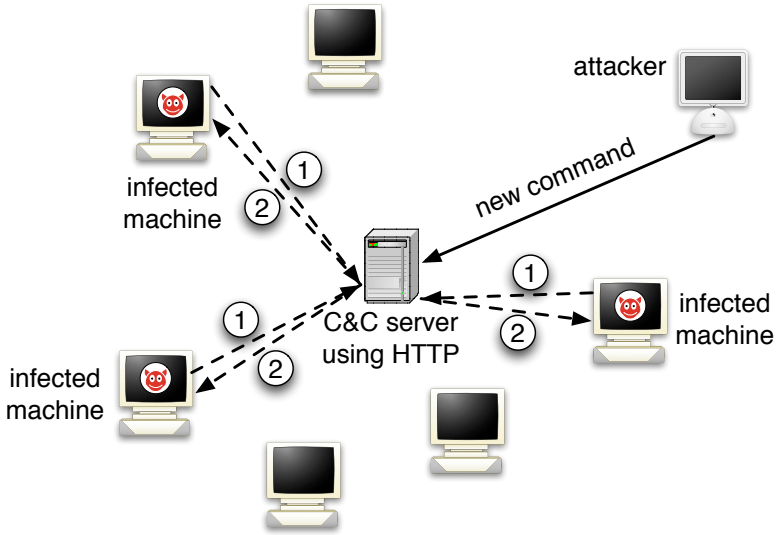


Figure 4.5: Communication flow in a HTTP-based botnet: the bots periodically (1) poll for new commands and (2) receive the commands as HTTP response by the C&C server.

botnets that use HTTP as communication protocol. This kind of botnets has one key characteristic that differentiates it from IRC-based botnets: instead of *pushing* the commands from the botmaster to the victims via IRC, such bots periodically *poll* the C&C server for new commands. Such a query can for example be:

(1) `/ger/rd.php?id=1-1C712F85E20BB42&ver=gr7`

(2) `/index.php?id=thonqfeoov&ver=19&cnt=DEU`

(3) `/px/new/notify.php?port=2721`

In the first example, the bot encodes in the query a randomly chosen ID and the version information about the bot. Similarly, in the second example the bot transmits within the query its randomly chosen ID, and information about the bot version and the country the bot is running in. Finally, in the third example, the bot just transmits a TCP port number back to the C&C server. At this TCP port, the bot has installed a listening tool that the attacker can use as a backdoor to access the system. As a reply to such a query, the C&C server sends a command back to the bot. Figure 4.5 illustrates the communication flow in an HTTP-based botnet. The query is periodically repeated by the bot, typically in a five minute or longer interval. The botnet controller can send a command to the C&C server and update the reply that is sent back to a bot's query. The next time a bot sends a request, the updated command is delivered to the bot.

The general method outlined in Chapter 3 can also be applied to this kind of remote control networks. With the help of honeypots, we can capture a propagating malware binary: in order to propagate further, a bot binary has to infect a victim and take control

of it. In order to be infected, we deploy honeypots as outlined in Section 4.4. Tools like Nepenthes, Amun, or GenIII honeypots are also useful to capture this kind of bots. With the help of an automated analysis as outlined in Section 4.5, we do not need any human intervention during the analysis process. CWSandbox enables us to automatically extract the network communication and thus detect the communication channel between the bot and the HTTP-based C&C server. Based on this information, we can then use botspy to monitor the C&C server: we periodically query the server, analyze the reply, and can thus track this kind of botnets.

When tracking HTTP-based botnets, we loose some of the insights compared to IRC-based botnets. We can still observe new commands launched by the attacker, but we can for example not estimate how many other bots are part of the HTTP-based botnet: since we cannot see how other victims send a request to the C&C server, we cannot count the number of victims.

4.8 Empirical Measurements

In this section we present some of the findings we obtained through our observation of botnets. Data is sanitized so that it does not allow one to draw any conclusions about specific attacks against a particular system, and it protects the identity and privacy of those involved. The information about specific attacks and compromised systems was forwarded to different CERTs (Computer Emergency Response Teams) which handle the incidents and inform the victims (if possible/necessary).

4.8.1 General Observations

The results are based on the observations collected with several virtual honeypot sensors, either running Nepenthes or a full high-interaction honeypot. We start with some statistics and informal observations about the botnets we have tracked in the measurement period between March and June 2007.

- *Number of botnets*: We were able to track more than 900 botnets during a four-month period. Some of them went offline (i.e., C&C server went offline) but typically about 450 active botnets were monitored.

- *Number of hosts*: During these few months, we saw more than 500,000 unique IP addresses joining at least one of the channels we monitored. Seeing an IP address means here that the C&C server was not modified to not send a JOIN message for each joining client. If an IRC server is modified not to show joining clients in a channel, we do not see IP addresses here. Furthermore, some IRC server obfuscate the joining client's IP address and obfuscated IP addresses do not count as seen, too. Note that we do not take churn effects like DHCP or NAT into account here. Nevertheless. this shows that the threat posed by botnets is high, given the large amount of infected machines. Even if we are very optimistic and estimate that we track a significant percentage of all botnets and all of our tracked botnet C&C

servers are not modified to hide JOINs or obfuscate the joining clients IPs, this would mean that a large number of hosts are compromised and can be controlled by malicious attackers.

- *Typical size of botnets*: Some botnets consist of only a few hundred bots. In contrast to this, we have also monitored several large botnets with up to 40,000 hosts. The actual size of such a large botnet is hard to estimate. Often the attackers use heavily modified IRC servers and the bots are spread across several C&C servers which are linked together to form a common remote control network.

- *Dimension of DDoS attacks*: We are able to make an educated guess about the current dimension of DDoS attacks caused by botnets. We can observe the commands issued by the controllers and thus see whenever the botnet is used for such attacks. During the observation period of four months, we were able to observe almost 300 DDoS attacks against 96 unique targets. Often these attacks targeted dial-up lines, but there are also attacks against bigger websites or other IRC servers.

- *Spreading of botnets*: Commands issued for further spreading of the bots are the most frequently observed messages. Commonly, Windows systems are exploited, and thus we see most traffic on typical Windows ports used for communication between Windows systems.

- *"Updates" within botnets*: We also observed updates of botnets quite frequently. Updating in this context means that the bots are instructed to download a piece of software from the Internet and then execute it. We could collect a little more than 300 new binaries by observing the control channels. These binaries commonly had a very low detection rate by antivirus engines.

- *Modified servers*: Something we also observe quite often is that the controllers change the protocol of the whole IRC server and modify it in such a way that it is not possible to use a traditional IRC client to connect to it. For example, the attacker can replace the normal IRC status messages and use other keywords. The modifications are often rather simple: A server can for example use SENDN and SENDU instead of the normal NICK and USER, respectively. But even this small change prohibits the use of a traditional IRC client to connect to this botnet and observe it. Due to the modular design of botspy, it is also easily possible to extend the tool and write a module that can communicate with the modified server.

- *Encryption*: There are also modifications regarding the communication protocol that we cannot easily adopt. For example, the botnet controller can implement an encryption scheme, i.e., she sends encrypted commands to the bots, which in turn decrypt and execute them. In such a case, the topic of the channel contains encrypted commands, which we cannot understand, unfortunately. By reverse engineering of the bot, it is possible to find out the issued command, but this is a time-consuming and cumbersome job. This indicates that encryption could pose

limitations for automation, especially if more and more botnets use encrypted command channels in the future.

Moreover, the data we captured while observing the botnets show that these control networks are used for more than just DDoS attacks. Possible usages of botnets can be categorized as listed here. And since a botnet is nothing more than a tool, there are most likely other potential uses that we have not listed.

- *Spamming*: Some bots offer the possibility to open a SOCKS v4/v5 proxy – a generic proxy protocol for TCP/IP-based networking applications – on a compromised machine. After enabling the SOCKS proxy, this machine can then be used for nefarious tasks such as sending bulk e-mail (*spam*) or phishing mails. With the help of a botnet and thousands of bots, an attacker is able to send massive amounts of spam. Some bots also implement a special function to harvest e-mail addresses from the victims.

  In addition, this can, of course, also be used to send phishing mails, since phishing is a special case of spam. Also increasing is so-called stock spam: advertising of stocks in spam e-mails. In a study we could show that stock spam indeed influences financial markets [BH06].

- *Spreading new malware*: In many cases, botnets are used to spread new bots. This is very easy, since all bots implement mechanisms to download and execute a file via HTTP or FTP. A botnet with 1,000 hosts that acts as the start base for the new malware enables very fast spreading and thus causes more harm. The Witty worm, which attacked the ICQ protocol parsing implementation in Internet Security Systems (ISS) products, could have been initially launched by a botnet because some of the attacking hosts were not running any ISS services and the number of initial infections was more than just one host [SM04].

- *Installing advertisement add-ons*: Botnets can also be used to gain financial advantages. This works by setting up a fake website with some advertisements. The operator of this website negotiates a deal with some hosting companies that pay for clicks on advertisements. With the help of a botnet, these clicks can be automated so that instantly a few thousand bots click on the pop-ups. This process can be further enhanced if the bot hijacks the start-page of a compromised machine so that the clicks are executed each time the victim uses the browser.

- *Keylogging*: If the compromised machine uses encrypted communication channels (e.g., HTTPs or POP3s), then just sniffing the network packets on the victim's computer is useless, since the appropriate key to decrypt the packets is missing. But most bots also implement functions to log keystrokes. With the help of a keylogger, it is very easy for an attacker to retrieve sensitive information.

- *Harvesting of information*: Sometimes we can also observe the harvesting of information from all compromised machines. With the help of special commands, the operator of the botnet can request sensitive information from all bots.

This list demonstrates that attackers can cause a great deal of harm or criminal activity with the help of botnets. With our method we can identify the root cause of all of these types of nuisances – namely the central command infrastructure – and hence the proposed methodology cannot only be used to combat DDoS.

4.8.2 Measurement Setup in University Environment

In the following, we present more precise measurements and analysis results of autonomous spreading malware activity within the network of RWTH Aachen University, Germany. With more than 40,000 computer-using people to support, this network offers us a testbed to study the effects of bots and worms. Our analysis is based on eight weeks of measurement, which took place during December 2006 and January 2007.

The network of RWTH Aachen university consists of three Class B network blocks (three /16 networks in CIDR notation). A Nepenthes sensor listens on about 16,000 IP addresses spread all across the network. Most of the IP addresses are grouped in a large block in one Class B network, but we have also taken care of evenly distributing smaller blocks of the sensor IPs all across the network to have a more distributed setup. This is achieved by routing smaller network blocks to the machine on which Nepenthes emulates the vulnerabilities. We do not need to install additional software on end-hosts, but use a purely network-based measurement approach.

4.8.3 Network-based Analysis Results

With the help of the logging modules of Nepenthes, we can keep track of all connections which were established to the sensor. That includes the attacker's IP address, the target IP address and port, as well as the vulnerability module which was triggered. More than 50 million TCP connections were established during the measurement period. Since the Nepenthes sensor is a *honeypot* and has no real value in the network, it should not receive any network connections at all. Thus, the vast majority of these 50 million network connections have a malicious source. On average, more than 900,000 TCP connections were established to the Nepenthes sensor per day and about 240,000 known exploits were performed every single day. Thus Nepenthes recognized about 27% of all incoming connection attempts as attacks and responded with a correct reply.

The remaining 73% of network connections are mainly caused by scanning attempts: about 75% of these connections target TCP port 80 and search for common vulnerable web applications. An additional 22% contain probe requests used by attackers during the reconnaissance phase in order to identify the network service running on a given target. About 3% of network connections contain a payload that Nepenthes could not understand. By manually adding support for these missed exploitation attempts, Nepenthes could be enhanced. With approaches like ScriptGen [LDM06], the detection rates could be automatically improved in the future as shown by preliminary results by Leita with SGNET [Lei08].

A total of more than 13,400,000 times the sensor system was *hit*. This means that this many times a TCP connection was established, the vulnerability emulation was

successful, and a malware binary could be downloaded by the sensor. If one host scans linearly the whole measurement range, then we count each of these connections as a separate one. Since this skews the data, we take only the unique hostile IP addresses into account. Roughly 18,340 unique IP addresses caused those hits. Table 4.1 depicts the sanitized IP addresses of the ten most active attackers together with the according country. It is evident that a small number of IP addresses are responsible for a signification amount of malicious network traffic.

IP Address:	**Country:**	**Hits:**
XXX.178.35.36	Serbia and Montenegro	216.790
XXX.211.83.142	Turkey	156.029
XXX.7.116.4	France	108.013
XXX.147.192.47	United States	107.381
XXX.92.35.23	Norway	94.974
XXX.206.128.27	United States	91.148
XXX.12.234.94	Japan	91.051
XXX.255.1.194	United States	78.455
XXX.92.35.24	Norway	78.439
XXX.29.103.225	United States	77.580

Table 4.1: Top ten attacking hosts with country of origin.

An analysis revealed that the distribution of attacking hosts follows a classical long-tail distribution as depicted in Figure 4.6. About 9,150 IP addresses, corresponding to about 50% of the total number observed, contacted the sensor system less then five times. These IP addresses are presumably infected with some kind of autonomous spreading malware which propagates further by scanning randomly for other victims.

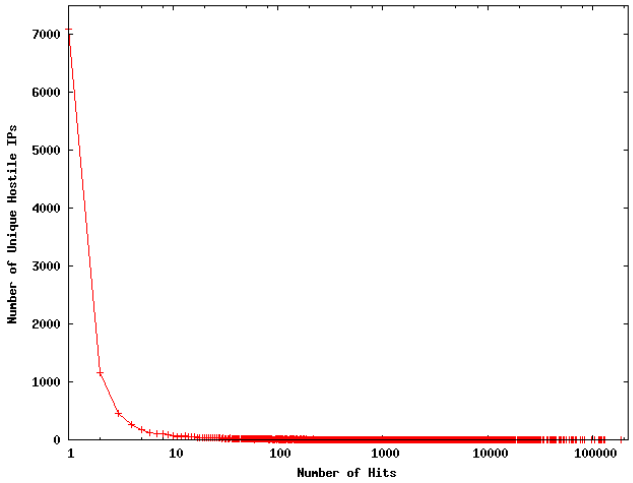


Figure 4.6: Distribution of attacking hosts.

TCP Port Number	Number
445	57,015,106
135	184,695
3127	23,076
80	20,746
42	18,653
139	15,112
1023	14,709
5554	13,880
6129	27
1025	1

(a) Top ten vulnerabilities detected by Nepenthes.

Dialogue	Number
LSASSDialogue	56,652,250
PNPDialogue	361,172
DCOMDialogue	184,696
SasserFTPDDialogue	28,589
MydoomDialogue	23,076
IISDialogue	20,746
WINSDialogue	18,655
NETDDEDialogue	15,112
SMBDialogue	2,341
DWDialogue	27

(b) Top ten TCP ports used by autonomous spreading malware.

Figure 4.7: Statistics of attacks observed with Nepenthes.

The 18,340 unique IP addresses we monitored during the analysis period connected to different TCP ports on the sensor. The distribution of target ports is very biased, with more than 97% targeting TCP port 445. This port is commonly used by autonomous spreading malware that exploits vulnerabilities on Windows-based systems. Table 4.7a provides an overview of the distribution.

Closely related to the distribution of target TCP ports is the type of vulnerabilities exploited. This distribution is also dominated by the most common vulnerability on TCP port 445: the `Lsasrv.dll` vulnerability, commonly referred to as *LSASS* (Microsoft Security Bulletin MS04-011 [Mic04]). Table 4.7b provides an overview of the vulnerability modules triggered and we see a bias towards the Windows vulnerabilities related to network shares.

The 13.4 million downloaded binaries turned out to be 2,454 unique samples. The uniqueness is determined by the MD5 hash of each binary: two binaries that have the same MD5 hash are considered to be the same binary. This is not foolproof due to the recent attacks on MD5 [WFLY, BCH06], but so far we have no evidence that the attacking community has released different binaries with the same MD5 hash. On the other hand, this is also no strong indicator for uniqueness: if the malware binary is polymorphic, i.e., it changes with each iteration, we collect many samples which in fact are very similar. In the middle of December 2006, such a polymorphic bot was released in the form of *Allaple* worm. In Section 4.8.4 we show preliminary results on how we can identify similar malware binaries based on behavior.

The number of collected samples result in an average of one unique malware binary every 5,240 hits. Considering the number of successful exploits per day, this results in almost 46 new binaries every 24 hours. Figure 4.8 provides an overview of the chronological sequence for the number of collected binaries and number of unique binaries. The number of collected binaries varies from day to day, ranging between

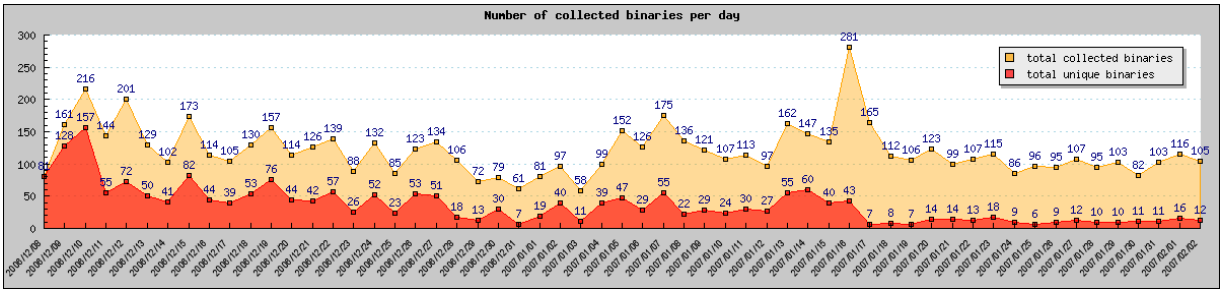


Figure 4.8: Chronological analysis of collected malware binaries.

58 and 281. There are several spikes in this measurement, but no reason for these anomalies could be identified with certainty. Presumably these spikes are caused by polymorphic samples. The situation is slightly different for the number of unique binaries: in the first few days, the number of unique binaries collected per day is high, whereas this number drops after about six weeks. It seems like there is some kind of *saturation*: in the beginning, the number of unique binaries is significantly higher than in the end of the measurement period. After a certain period of time we have collected the commonly propagating malware in the measurement network and only a few new binaries are collected per day. This number varies between 6 and 16, an educated guess would be that this corresponds to new malware binaries released by attackers on a daily basis that hit our sensor.

4.8.4 CWSandbox Analysis Results

In this section, we present some quantitative statistics about the analysis results of our malware collection. It should be mentioned that our collection cannot give a representative overview of current malware on the whole Internet, as on the one hand, the sample set size is not large enough and, on the other hand, it contains *only* autonomous spreading applications like bots and worms. However, for this particular subset of malicious activity, our measurement setup can provide us with an overview of the current threat level for a typical university environment.

From the overall collected 2,454 sample files, 2,034 could be analyzed correctly by CWSandbox, one failed due to a crash and 419 were no valid Win32 applications. This means that in roughly 17% of the collected samples, the resulting file was not a valid executable. This can be explained by aborted transfers or disrupted network connectivity. One additional file of the remaining 2034 had a valid PE header, but could not be correctly initialized by the Windows Loader due to an access violation exception. Each successful analysis resulted in an XML analysis report, which reflects all the security-relevant operations performed by the particular file. As we are not interested in a detailed malware analysis for single file instances, we present quantitative results extracted from the 2,034 valid reports. The main focus of our statistics lies on network activities, but a few other important results are presented as well.

Remote TCP port	# samples
445	1312
80	821
139	582
3127	527
6667	403
6659	346
65520	143
7000	30
8888	28
443	16

Table 4.2: Top ten outgoing TCP ports used.

1,993 of the 2,034 valid malware samples tried to establish some form of TCP/IP connection, either outgoing, incoming (i.e., listening connections) or both. Besides DNS requests, we have not found any single malware in our set that uses only UDP directly. 1,216 binaries were successful in the attempt to setup an outgoing TCP connection. For all the others, the remote host was not reachable or refused the connection for some other reason. Altogether, 873 different TCP remote ports have been used for outbound connection attempts, and Table 4.2 shows the top ten of them. It is highly probable that most connections on port 445 and 139 are aiming on further malware propagation, port 80 and 443 are used for HTTP(s) connections, 6667, 66520, 7000, 6659, and 8888 are used for IRC communication, and port 3127 is presumably used as a backdoor by the malware family *MyDoom*.

Furthermore, we have found 1,297 samples that install a TCP server for incoming connections, most of them setting up an Ident server on port 113 for supporting IRC connections (497 samples). Other common ports on which malware opens TCP servers is port 3067 (122 samples), port 80 (9 samples), and port 5554 (7 samples).

Since most bots rely on IRC communication, a deeper investigation of these connections is necessary. 505 samples could successfully establish a connection to an IRC server. Moreover, 352 files tried to send IRC commands over an *unestablished* connection. The reason for that is most probably bad software design. 349 of these files are variants of *GhostBot*, the other 3 are *Korgo* worms. Furthermore, we have 96 samples that try to connect to a remote host on TCP port 6667 or 7000 and fail. Adding these numbers, we have at least 953 files which try or are successful in setting up an IRC connection. The corresponding samples are most probably IRC bots. We cannot know, how many of these different binaries belong to the same bot variant or even to the same botnet. However, by taking the remote IP address, remote TCP port, IRC channel name and IRC channel password into account, we can give an educated guess since this quadruple is a good indication for uniqueness: if a given binary uses the same tuple of network parameters, we can be sure that it is the same variant, although the MD5 sum of these binaries is different. Of all established IRC connections, 64 different *host:port:channel:channelpassword*-combinations have been used. As the host IP address for a botnet may change, we generalize the results to the different *chan-*

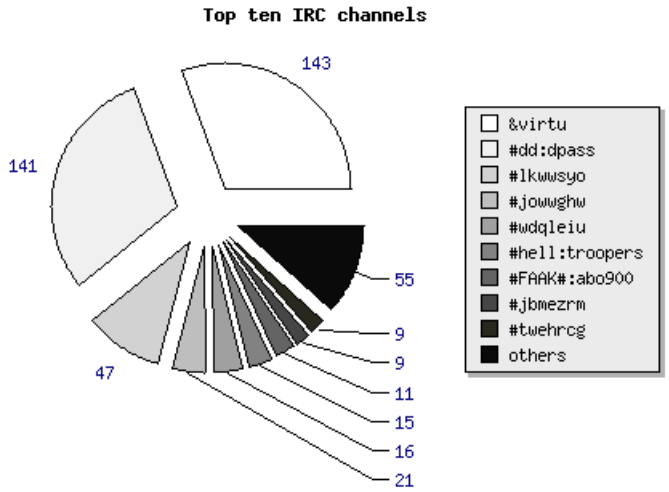


Figure 4.9: Distribution of IRC channel:password combinations.

nel:channelpassword-combinations and assume that each of those represent a different botnet or at least a different bot family. By generalizing the number of different combinations decreases down to 41. The most common channels are *&virtu (no password)* and *dd (password "dpass")* with 143 and 141 samples, respectively. These samples have a different MD5 sum, but based on their network behavior we argue that they are very similar. An overview of these results is given in Figure 4.9. Please note that all the combinations with only one corresponding malware sample are aggregated into *others*. We have also developed some techniques to classify a given set of malware samples based on their behavior [RHW+08], but refrain from detailing these results since it is beyond the scope of this section.

When looking at the different remote TCP ports which are used for establishing an IRC connection, we see that not only the default IRC port 6667 is used, but many more. It is interesting to see that, beside some probably random ports, a lot of well known ports of other well-known protocols are used, e.g., port 80 (HTTP), port 443 (HTTPS) or port 1863 (Microsoft Notification Protocol, used by a number of Instant Messaging clients). This allows the bot to communicate through a firewall that is open for these standard protocols. Thus it is necessary to also closely observe these port when thinking about vulnerability assessment. Figure 4.10 shows a distribution diagram of the TCP ports observed during the study.

As already mentioned above, a few other interesting, system-level related results can be drawn from our analysis reports as well. After infecting a new host, nearly all malware samples try to install some auto-start mechanism, such that it is activated each time the infected system reboots. This is commonly done by adding some auto-start registry keys, but some malware install a *Windows Service Application* or even a *kernel driver*. By doing that, it is much harder to detect the presence of malware. Especially

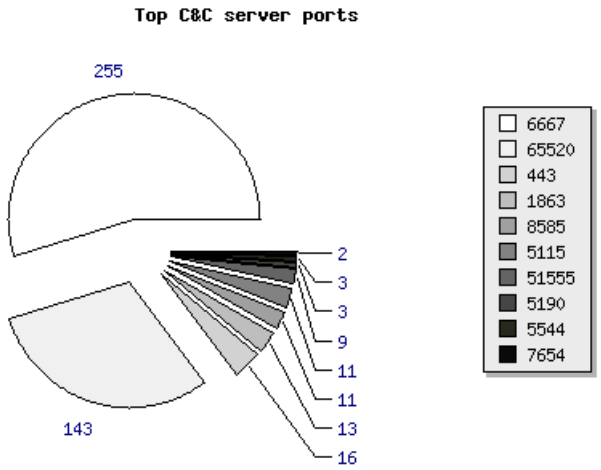


Figure 4.10: TCP ports used for IRC connections.

Servicename	**Filename** (base directory is C:\Windows\)	**Kernel driver**	**# Samples**
SVKP	`system32\SVKP.sys`	x	15
DLLHOST32	`system\dllhost.exe`		8
WINHOST32	`system\services.exe`		2
Print Spooler	`system32\spooler.exe`		1
hwclock	`system32\hwclock.exe`		1
oreans32	`system32\drivers\oreans32.sys`	x	1
Windows System 32	`services.exe`	x	1
Windows Terminal Services	`system32\vcmon.exe`		1
Advanced Windows Tray	`system32\vcmon.exe`		1
Windows MSN	`wmsnlivexp.exe`		1
Windows Process Manager	`system32\spoolsc.exe`		1
mside	`system\mside.exe`		1
TCP Monitor Manager	`system32\symon.exe`		1
Client Disk Manager	`system32\symon.exe`		1
Monitor Disk Manager	`system32\spoolcs.exe`		1
System Restore Manager	`system32\symon.exe`		1

Table 4.3: Services and kernel drivers installed by collected malware samples.

in the case of kernel drivers, the malware binaries can get higher security privileges on the local system. Table 4.3 provides an overview of the services and kernel drivers installed by the samples we collected. In total, 21 of our collected files install a service application and 17 install a kernel driver. Since some of these binaries use the same service name and filename for the given processes, we can learn that these were most probably installed by variants of the same malware family.

As a final statistic result, Table 4.4 shows a summary of the windows processes, into which the malware samples injected malicious code. It is a common approach to

Injection target process (base directory is C:\Windows\)	**# Samples**
`explorer.exe`	787
`system32\winlogon.exe` **and** `explorer.exe`	101
`system32\winlogon.exe`	74

Table 4.4: Injection target processes observed for collected malware samples.

create a new thread (or modify an existing one) in an unsuspicious windows process, e.g., `explorer.exe` or `winlogon.exe`, and perform all malicious operations from that thread. By doing this, the malware becomes more stealthy and, furthermore, circumvents local firewalls or other security software that allows network connections only for trusted applications.

4.8.5 Antivirus Engines Detection Rates

In order to evaluate the performance of current antivirus (AV) engines, we scanned all 2,034 binaries, which we had captured with Nepenthes and successfully analyzed with CWSandbox, with four common AV engines. This scan was performed one week after the measurement period, in order to give AV vendors some time to develop signatures and incorporate them into their products. This test helps us to estimate the detection rate for common autonomous spreading malware and also for vulnerability assessment. These binaries are currently spreading in the wild, exploited a vulnerability in our measurement system, and could be successfully captured. In contrast to common antivirus engine evaluation tests, which partially rely on artificial test sets, this set represents malware successfully spreading in the wild during the measurement period.

AV software	**Absolute detection**	**Relative detection rate**
AntiVir	2015	99.07%
ClamAV	1963	96.51%
BitDefender	1864	91.64%
Sophos	1790	88.00%

Table 4.5: Detection rates for 2,034 malware binaries for different AV scanners.

Table 4.5 displays the current detection rates of each scanner with the latest signature version installed. Nevertheless, none of the tools was able to detect all malicious files and classify them accordingly. The malware reports vary significantly from one tool to another. Figure 4.11 provides an overview of the detected malware binaries by the four different antivirus engines.

We focus on the ClamAV results in the following and analyze the different malware families more closely. ClamAV detected 137 different *malware variants* in the test set of 2,034 samples. In total, 27 different *families* of malware could be identified. Table 4.6 provides an overview of the top ten different malware variants. Two families of malware clearly dominate the result: *Padobot* and *Gobot* are the two main autonomous spreading malware families we could observe within our measurement environment. Besides these

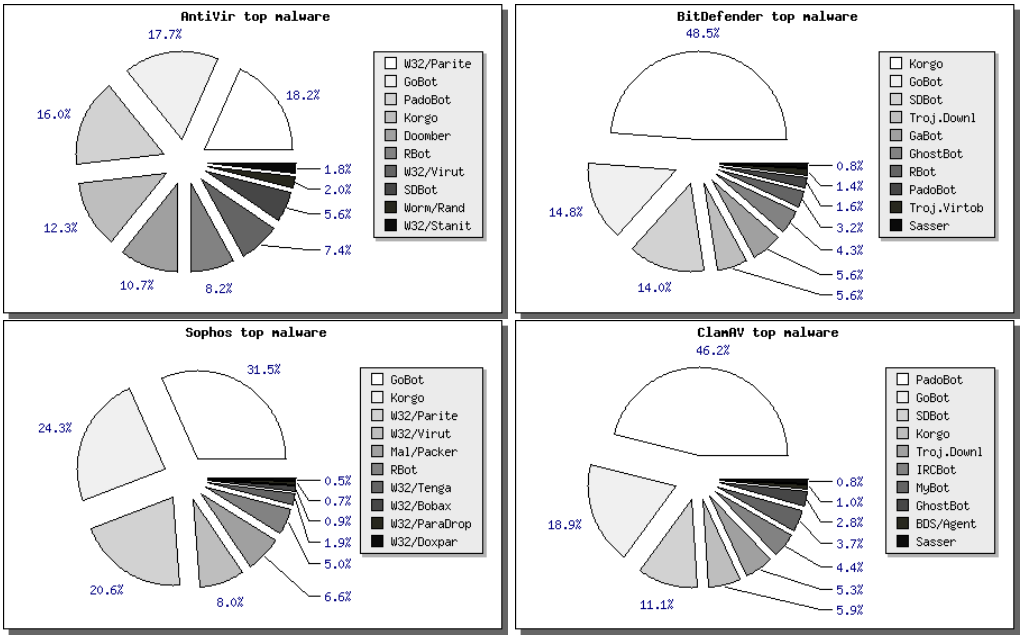


Figure 4.11: Malware variants detected by different antivirus engines.

Malware Variant	Number of Samples
Worm.Padobot.M	426
Worm.Padobot.P	274
Trojan.Gobot-3	118
Trojan.Gobot-4	106
Worm.Padobot.N	101
Trojan.Downloader.Delf-35	100
Trojan.IRCBot-16	76
Trojan.Gobot.A	61
Trojan.Ghostbot.A	53
Trojan.Gobot.T	37

Table 4.6: Top ten different malware variants.

two families, also many other forms of autonomous spreading malware are currently spreading in the wild. Although Padobot and Gobot dominate the list of malware variants, the largest number of different variants was captured for *SdBot*: 35 different variants of the same family could be captured, whereas Padobot (14) and Gobot (8) had significantly less different variants.

Some of the malware binaries are already known for a long time. For example, the first variants of *Blaster* were observed in the wild in August 2003. More than three years later, we captured four different variants of Blaster which are still propagating on the

Internet. Similarly, three different variants of *Sasser*, which first appeared in April 2004, were captured during the measurement period. We conclude that there are still systems on the Internet which are infected for a long time, helping "old" malware binaries to propagate further even today.

4.8.6 Botspy Analysis Results

With the help of botspy, we can study the remote control networks used by the autonomous spreading malware: in case we have captured a bot, it connects to this remote control network so that the attacker can send it commands. With the help of botspy, we can infiltrate this remote control network and observe from the inside what is happening within the botnet.

In total, we could observe 41 different botnet Command & Control (C&C) servers. Again, this behavior can be used to classify malware samples: if two binaries connect to the same C&C server, we can argue that they are similar despite the fact that they have a different MD5 sum. For example, 149 binaries connected to the IP address XXX.174.8.243 (`home.najd.us`). The system-level behavior of these samples is also very similar, so presumably these are just minor variants of the same malware family.

When connecting to the botnets, we could observe 33 different topics in the channel used to command the bots. The most common command used by the botmaster was related to propagation: most bots are instructed to scan for other vulnerable machines and exploit vulnerabilities on these systems. More and more common are botnets that use non-standard IRC or encrypted communication mechanisms. For example, the command sent by the botmaster to the bots could be an encrypted string. In total, we found 10 different botnets that use encrypted, IRC-based communication. Without a proper decryption routine, it is hard to study this kind of botnets. Currently it is unclear how we can efficiently study this kind of botnets, since only manual reverse engineering can uncover the current command issued by the botmaster.

Estimating the size of a given botnet is a hard task [RZMT07]. One possibility to estimate the size is to rely on the statistics reported by the C&C server upon connect: if the IRC server is not configured properly, it reports the number of connected clients. Moreover, we can query the server for the number of connected clients and various other status messages. Based on these numbers, the typical botnet size in our sample set varied between only a few hundred up to almost 10,000 bots.

Besides the IRC-based bots, we also found several samples of HTTP-based bots. These bots periodically query a given HTTP server and the response contains commands which are executed by the bot. Due to the rather stealthy communication channel of such bots, detection becomes harder. In addition, measuring the size of such a botnet is hard since we can only passively monitor the HTTP server. In total, we could identify seven different botnets that use HTTP-based communication.

4.9 Mitigation

Several ways to prevent DDoS attacks caused by botnets exist that we want to sketch in this section. Since we observe the communication flow within the botnet, we are also able to observe the IP addresses of the bots unless this information is obfuscated, e.g., by modifying the C&C server. Thus one possible way to stop DDoS attacks with this methodology is to contact the owner of the compromised system. This is however a tedious and cumbersome job that does not scale since many organizations are involved and these organizations are spread all over the world. In addition, the large number of bots make this approach nearly infeasible, only an automated notification system could help. For now it is unclear how such a system could be implemented and operated in an efficient and scalable manner.

Another approach to prevent DDoS attacks caused by botnets aims at stopping the actual infrastructure, in particular the C&C server, since this component is vital for the remote control network. Currently, the most effective method to stop bots is to stop the initial establishment of a connection from a bot to the C&C server. The botnets we analyzed in this chapter use a central server for Command and Control (commonly either based on the protocol IRC or HTTP), and, in most cases, a dynamic DNS name is used for this server. This allows us to stop a botnet effectively: once we know this DNS name, we can contact the DNS provider and file an abuse complaint. Since many DNS providers do not tolerate abuse of their service, they are also interested in stopping the attack. The DNS provider can easily *blackhole* the dynamic DNS name, i.e., set it to an IP address in the private range as defined in RFC 1918. If an infected machine then tries to contact the C&C server, the DNS name will resolve to a private IP address and thus the bot will not be able to contact the C&C server. This method is commonly used by CERTs and similar organizations and has proven to be quite effective; many communication channels have been disrupted in this way. Nevertheless, it requires the DNS provider's cooperation, which is not always the case.

There are also several methods to stop a bot within a network that can be carried out by a network administrator or security engineer. We discuss several methods in the following. As always, the best way to cancel a threat is to stop its root cause. In this case, this would mean eliminating the attack vectors and checking for signs of intrusions, e.g., by patching all machines and keeping AV signatures up-to-date. But this is often difficult: a zero-day exploit cannot be eliminated in all cases, and patching needs some testing since it could break important systems. In addition, AV scanners often cannot identify targeted attacks. In several recent incidents, the time between a proof-of-concept exploit for a new security vulnerability and the integration of it into a bot can be as little as several hours or days, so patching cannot always help; nevertheless, it is still important to try to keep patches as up to date as possible.

One quite effective method to detect the presence of bots also exploits their rather noisy nature. Most bots try to spread by exploiting security flaws on other systems. To find such a system, they have to extensively scan the network for other machines. In addition, the communication channel often uses specific, rather unusual ports. Thus by looking at the state of the network, it is often possible to detect bots. Flow-based

approaches like *Netflow*/*cflow* are easy-to-use solutions for this problem, in which the collected data often allows an administrator to spot an infected machine. A typical sign is a spike in the number of outgoing connections, most often on TCP ports 445 and 135, or on ports with recent security vulnerabilities, caused by bots that try to propagate via common vulnerabilities. Another sign is a high amount of traffic on rather unusual ports. We analyzed the information about more than 1,400 botnets and found out that the vast majority of botnets use TCP port 6667 for C&C. Other commonly used ports include TCP ports 7000, 3267, 5555, 4367, and 80. TCP port 6667 is commonly used for IRC, and of course 80 for HTTP, but an administrator should take a look at these and the others mentioned.

We developed simple, yet effective bot-detection system that relies on detecting the communication channel between bot and C&C server [GH07]. The technique is mainly based on passively monitoring network traffic for unusual or suspicious IRC nicknames, IRC servers, and uncommon server ports. By using n-gram analysis and a scoring system, we are able to detect bots that use uncommon communication channels, which are commonly not detected by classical intrusion detection systems. Upon detection, it is possible to determine the IP address of the C&C server, as well as, the channels a bot joined and the additional parameters which were set. The software *Rishi* implements the mentioned features and is able to automatically generate warning emails to report infected machines to an administrator. Within the 10 GBit network of RWTH Aachen university, we detected 82 bot-infected machines within two weeks, some of them using communication channels not picked up by other intrusion detection systems. Of course, such a method requires some human supervision, since it is not error-free and could lead to false positives. In addition, the C&C commands can change with time, and thus regular updates are necessary. There exist many other botnet detection systems that try to detect different aspects of botnet communication [GPY+07, GZL08, GPZL08, YR08].

4.10 Summary

Today, botnets are a problem for individuals and also corporate environments. Due to their immense size (several thousand compromised systems can be linked together), botnets pose a severe threat to the Internet community: they are often used for DDoS attacks, to send spam or phishing mails, and as spyware to steal sensitive information from the victim's machine. Since an attacker can install programs of her choice on the compromised machines, her procedures are arbitrary.

In this chapter, we have exemplified a technical realization of the methodology proposed in Chapter 3 to prevent malicious remote control networks considering as example the tracking of botnets with a central server that is used for Command and Control. We showed how to use honeypots to collect more information related to a botnet. With the help of low-interaction honeypots that emulate vulnerabilities in network services, we can capture samples of autonomous spreading malware in an automated way. By analyzing the collected samples with CWSandbox, we obtain — also in an automated way — a behavior-based report that contains amongst other

information valuable data about the botnet itself. Based on this information, we can infiltrate the botnet and observe it from the inside based on a light-weight agent. With the help of the collected information, we can then try to mitigate the threat, e.g., by contacting the DNS provider to withdraw the domain name related to the botnet. The important point here is that we are able to automate most of the collection, analysis, and infiltration steps to a high degree based on the tools we introduced in this chapter. Since botnets are an automated threat, we also need an automated countermeasure.

More research is needed in this area. Current botnets are rather easy to stop due to their central C&C server. But in the future, we expect other communication channels to become more relevant, especially peer-to-peer-based C&C communication. We have seen the first bots that use such communication channels with Sinit [Gro03], Nugache [Naz06], and Storm Worm [Ste07], but presumably the future will bring many more of these types of malware. Thus the next chapter focusses on botnets with a peer-to-peer-based Command and Control infrastructure and we show how the methodology we introduced in Chapter 3 can also be used to mitigate these botnets.

Chapter 5

Tracking Botnets with Peer-to-Peer-based C&C Server

5.1 Introduction

Today we are encountering a new generation of botnets that use peer-to-peer style communication. These botnets do not have a central server that distributes commands as we have analyzed in the previous chapter, but the botnets rely on peer-to-peer communication to disseminate commands to the bots. Simply shutting down the central server that is used to disperse the commands is thus not feasible since there is no central server. We thus need another way to stop this kind of botnets. As a second case study on the feasibility of our methodology to track malicious remote control networks introduced in Chapter 3, we focus in this chapter on botnets with a peer-to-peer-based communication channel and show that we can also successfully track these botnets with the proposed methodology. As a running example, we analyze *Storm Worm*, one of the most prevalent botnets observed so far.

Our measurements show that our strategy can be used as a way to disable the communication within the Storm Worm botnet to a large extent. As a side effect, we are able to estimate the size of this botnet, in general a hard task [RZMT07]. Our measurements are much more precise than previous measurements [GSN+07, Kre07]. This is because previous measurements were based on *passive* techniques, e.g., by observing visible network events like the number of spam mails supposedly sent via the bots. We are the first to introduce an *active* measurement technique to actually enumerate the number of infected machines: we crawl the peer-to-peer network, keep track of all peers, and distinguish an infected peer from a regular one based on the characteristic behavior of a bot.

Contributions. To summarize, our work presented in this chapter makes the following three main contributions:

1. We show how the method of tracking malicious remote control networks as introduced in Chapter 3 can be used to track peer-to-peer based botnets. We argue that the method is applicable to analyze and mitigate *any* botnet using peer-to-peer publish/subscribe-style communication.

2. We demonstrate the applicability by performing a case study of Storm Worm, thereby being the first to develop ways to mitigate the Storm Worm botnet.

3. In doing this, we present the first empirical study of peer-to-peer botnets giving details about their propagation phase, their malicious activities, and other features.

Outline. This chapter is structured as follows: we first motivate our work by introducing botnets that use peer-to-peer based communication channels in Section 5.2. In Section 5.3 we show how the methodology proposed in Chapter 3 can be applied to this kind of botnets. A technical overview of Storm Worm, our running example, is given in Section 5.4. In Section 5.5 we show how we can actually track the Storm Worm botnet, before we present in Section 5.6 empirical measurement results. Section 5.7 discusses how to mitigate this botnet and we finally conclude in Section 5.8.

5.2 Motivation

As introduced in the previous chapter, the common control infrastructure of botnets in the past was based on Internet Relay Chat (IRC): the attacker sets up an IRC server and opens a specific channel in which she posts her commands. Bots connect to this channel and act upon the commands they receive from the botmaster. Today, the standard technique to mitigate IRC-based botnets is called *botnet tracking* and the whole method was discussed in depth in Section 4.7.1. To summarize, botnet tracking includes three steps: the first step consists of acquiring and analyzing a copy of a bot in an automated way. This can be achieved for example using honeypots [BKH+06] and special analysis software [BMKK06, WHF07]. In the second step, the botnet is infiltrated by connecting to the IRC channel with a specially crafted IRC client [Ove07]. Using the collected information, it is possible to analyze the means and techniques used within the botnet. More specifically, it is possible to identify the central IRC server which, in the third and final step, can be taken offline by law enforcement or other means [Fed07]. An attacker can also use an HTTP server for distributing commands: in this setup, the bots periodically poll this server for new commands and act upon them. The botnet tracking methodology proposed in this thesis can also be applied in such a scenario as we have shown in Section 4.7.2.

Today we are encountering a new generation of botnets that use peer-to-peer style communication. These botnets do not have a central server that distributes commands and are therefore not directly affected by botnet tracking. Probably the most prominent peer-to-peer bot currently spreading in the wild is known as *Peacomm*, *Nuwar*, or *Zhelatin*. Because of its devastating success, this worm received major press coverage [Gro07, Kre07, Nau07] in which – due to the circumstances of its spreading – it was given the

name *Storm Worm* (or *Storm* for short) [Ste07]. This malware is currently the most wide-spread peer-to-peer bot observed in the wild. Other botnets that use peer-to-peer techniques include Sinit [Gro03] and Nugache [Naz06]. Due to their success and resistance against take-down, we expect many more such botnets appearing in the wild in the near future.

In this chapter we show how the technique of botnet tracking can be extended to analyze and mitigate peer-to-peer based botnets. Roughly speaking, we adapt the three steps of botnet tracking in the following way using Storm Worm as a case study: in the first step, we must get hold of a copy of the bot binary. In the case of this botnet, we use spam traps to collect Storm-generated spam and client-side honeypots to simulate the infection process. This is similar to the approach in the previous chapter in which we used low-interaction honeypots that emulate vulnerabilities in network services. The second step, the infiltration of the botnet, is adopted since we need to use a peer-to-peer protocol instead of IRC, HTTP, or other client/server protocols. From a methodological point of view, only the protocol used during the infiltration phase changes, the overall method stays the same. The third step, the actual mitigation, is the most difficult: in the case of Storm Worm we exploit weaknesses in the protocol used by the bot to inject our own content into the botnet, in an effort to disrupt the communication between the bots. We argue later that this method is effective against any peer-to-peer botnet using content-based publish/subscribe-style communication, as we now explain.

5.3 Botnet Tracking for Peer-to-Peer-based Botnets

We now present a general method to analyze and mitigate specific peer-to-peer botnets.

5.3.1 Class of Botnets Considered

The class of botnets we consider in this chapter are those which use unauthenticated content-based publish/subscribe style communication. This communication paradigm is popular in many of the well-known file sharing systems like Gnutella, eMule, or BitTorrent. The characteristics of such systems can be summarized as follows:

- *Peer-to-peer network architecture*: These networks have in common that all network nodes are both clients and servers. Any node can provide and retrieve information at the same time. This feature makes peer-to-peer networks extremely robust against node failures, i.e., they provide *high resilience*. Moreover, a peer-to-peer network has no central server that can be taken down in order to mitigate the whole network.

- *Content-based publish/subscribe-style communication*: In such systems the network nodes do not directly *send* information to each other. Instead, an information provider *publishes* a piece of information i, e.g., a file, using an identifier which is derived solely from i. An information consumer can then *subscribe* to certain information using a filter on such identifiers. In practice, such identifiers can be

derived from specific content of i or simply computed using a hash function. The peer-to-peer system matches published information items to subscriptions and delivers the requested information to the consumer. This is the common model implemented by most file sharing peer-to-peer systems, where each participant publishes his content and other participants can search for specific content using an identifier.

- *Unauthenticated communication*: Content providers do not authenticate information, but authentication is usually implicit: if the information received by a peer matches its subscription, then it is assumed to be correct. None of the popular file sharing systems provides any authentication procedure.

Note that in such systems communication is very *loosely coupled*. Neither information consumers know in general, which node published the information they receive, nor does an information provider know, which nodes will receive their published information. Both points, loose coupling and high resilience, make these networks attractive technologies for running botnets.

5.3.2 Botnet Tracking for Peer-to-Peer Botnets

We now introduce a widely applicable method to analyze and mitigate any member of the class of botnets described above. We show how the botnet tracking method introduced in Chapter 3 can be adopted for botnets that use peer-to-peer networks and exemplify the method with the help of a case study on the Storm Worm botnet later on.

Step 1: Exploiting the peer-to-peer Bootstrapping Process. A bot spreading in the wild must contain information to bootstrap itself within the botnet. In the case of peer-to-peer botnets, the bot *must contain* sufficient information on how to *connect* to the botnet and how to *receive* commands from the attacker. Usually this information includes a number of IP addresses of initial peers, service ports and application-specific connection information. By getting hold of and analyzing a bot, it is possible to extract this information by either active or passive means.

Getting hold of a bot means to simulate the infection process, which is a technique known from the area of honeypot technology. The main difficulties here are (1) to find out the infection vector and (2) to simulate vulnerable applications. While (1) may take some time and is hard to automate, (2) can be efficiently automated, e.g., using sandbox or network analysis techniques. The result of this step is a list of network locations (IP address/port) of peer services that form part of the peer-to-peer botnet.

This step is similar to the method we introduced in Section 4.4 and 4.5 which relies on honeypots to capture bots and sandboxes to automatically analyze the samples.

Step 2: Infiltration and Analysis. As a result of step 1, we also retrieve connection information to actually *join* the botnet. Joining the botnet means to be able to receive botnet commands issued by the attacker. By crafting a specific peer-to-peer client,

infiltration of the botnet remains a dangerous, but technically manageable process. It can be dangerous since the attacker could notice the infiltration process and start to specifically attack us.

This step is similar to the infiltration phase of botnets with a central server as outlined in Section 4.6.

Step 3: Mitigation. The mitigation of botnets must attack the control infrastructure to be effective, i.e., either the servers or the communication method. We now argue that publish/subscribe-style communication has weaknesses which can be generally exploited. In a botnet, the attacker wishes to in some way send commands to the bots. This is the characteristic of remote control. However, in publish/subscribe systems, there is no way to send information directly. Instead, a broadcast is simulated, as we now explain. The attacker defines a set $C = \{c_1, c_2, \ldots\}$ of botnet commands. At any point in time, whenever she wishes to send a command c_i to the bots, she publishes c_i in the peer-to-peer system. The bots must be able to receive all the commands from the attacker so they subscribe to the entire set C and can then accept commands.

Note that since we consider *unauthenticated* publish/subscribe systems, any member of the peer-to-peer system can publish c_i. This is the idea of our mitigation strategy: using the client from step 2, we can now try to either inject commands into the botnet or disrupt the communication channel. In general, disruption is possible: we can flood the network with publication requests and thus "overwrite" publications by the attacker. In order to actually inject commands, we need to understand the communication process in detail and then publish a specially crafted c_i.

This step is in principle similar to the mitigation methods we introduced Section 4.9 for botnets with a central C&C server, but requires a different implementation.

5.4 Technical Background

Before exemplifying our methodology of tracking peer-to-peer botnets, we provide an overview of Storm Worm. Please note that this description is a summary of the behavior we observed when monitoring the Storm botnet for a period of several months during August 2007 and June 2008. The attackers behind this network quite frequently change their tactics and move to new attack vectors, change the communication protocol, or change their behavior in other ways. The results from this section describe several important aspects of Storm and we try to generalize our findings as much as possible. Together with the work by Porras et al. [PSY07] and Kreibich et al. [KKL+08b], this is at the time of writing the most complete overview of Storm Worm.

5.4.1 Propagation Mechanism

A common mechanism for autonomous spreading malware to propagate further is to exploit remote code execution vulnerabilities in network services. If the exploit is successful, the malware transfers a copy of itself to the victim's machine and executes this

copy in order to propagate from one machine to another. This propagation mechanism is used for example by CodeRed [MSkc02], Slammer [MPS+03], and all common IRC bots [BY07]. Storm Worm, however, propagates solely by using e-mail, similar to mail worms like Loveletter/ILOVEYOU and Bagle. The e-mail body contains a varying English text that tries to trick the recipient into either opening an attachment or clicking on an embedded link. The text uses social engineering techniques in order to pretend to be a legitimate e-mail, e.g., we found many e-mails related to Storm Worm that feign to be a greeting card in order to trick the victim into clicking the embedded link.

With the help of *spamtraps*, i.e., e-mail addresses not used for communication but to lure spam e-mails, we can analyze the different spam campaigns used by Storm Worm for propagation. We have access to a spamtrap archive between September 2006 and September 2007 which receives between 2,200 and 23,900 spam messages per day (8,500 on average). The first Storm-related message we are aware of was received on December 29, 2006: it contained best wishes for the year 2007 and as an attachment a copy of the Storm Worm binary. An analysis of this archive shows that this botnet is quite active and can generate a significant amount of spam: we found that the botnet was in some period responsible for more than 10% of all spam messages received in this particular spamtrap.

The attackers behind Storm change the social engineering theme quite often and adopt to news or events of public interest. For example, the name "Storm Worm" itself relates to the subject used in propagation mails during January 2007 which references the storm Kyrill, a major windstorm in Europe at that time. For events of public interest (e.g., Labor Day, Christmas, start of NFL season, or public holidays), the attackers use a specific social engineering scam. Furthermore, they also use general themes (e.g., privacy concerns or free games) to trick users into opening the link in the e-mail message. In total, we counted more than 26 different e-mail campaigns for the period between December 2006 and June 2008.

The different campaigns also reveal a growing sophistication in the propagation mechanism. The first versions of Storm used an e-mail attachment to distribute a binary copy of the malware and tried to trick the recipient into opening the attachment, thus infecting the machine. However, in May/June 2007, the attackers changed their tactics and began to include a link to a malicious Web site in the e-mail body.

This change has presumably two main reasons: first, by not including the actual binary in the mail, there is no malicious attachment that an antivirus engine at the e-mail gateway could detect. Thus the chances are higher that the e-mail is not filtered and actually reaches the intended recipient. Second, if the recipient clicks on the link, a Web site opens which contains several exploits for vulnerabilities in common web browsers. Presumably the attackers hope that the victim has not patched the browser and is thus vulnerable to this kind of attacks.

To study the next step in the propagation phase, we examined the links from Storm-related e-mails with the help of *honeyclients*. A honeyclient is a system designed to study attacks against client applications, in our case attacks against a web browser [WBJ+06]. We implemented our own client-side honeypot which can be used to analyze a given Web site with different kinds of browsers on top of CWSandbox [WHF07]. Based on this

system and the analysis report generated by CWSandbox, we can determine whether or not the visited site compromised our honeypot. During five of the different spam campaigns we examined several URLs referenced in the e-mails. We used different releases of three web browsers, resulting in a total of eight different browser versions. The results indicate that Storm exploits only web browsers with a specific `User-Agent`, a HTTP request header field specifying the browser version. If this header field specifies a non-vulnerable browser, the malicious server does not send the exploit to the client. However, if the client seems to be vulnerable, the server sends between three and six different exploits for vulnerabilities commonly found in this browser or in common browser-addons. The goal of all these exploits is to install a copy of the Storm Worm binary on the visitor's machine. We observed that the actual exploit used in the malicious Web sites is polymorphic, i.e., the exploit code changes periodically, in this case every minute, which complicates signature-based detection of these malicious sites.

If the malicious Web site successfully compromises the visitor's web browser or the visitor falls for the social engineering scam and intentionally installs the binary, the victim is infected. The binary itself also shows signs of polymorphism: when continuously downloading the same binary from the same web server, the size (and accordingly the MD5 checksum) changes every minute. An analysis revealed that the changes are caused by periodically re-packing the binary with an executable packer which is responsible for the change in size.

In total, we collected more than 7,300 unique binaries over several weeks in August and September 2007. We tested this malware corpus with common antivirus engines and found that they have rather good detection rates for these samples because between 82 and 100 percent were detected. At least six different variants of the binary were used by the attackers, with a size of the binary between 97 and 164 KB. The different variants contain the same basic functionality, however the attackers slightly change each variant, e.g., by including code to detect virtual machines [PSY07]. We can also use the polymorphism observed for the binaries to collect evidence that the webservers are in fact only proxies which redirect requests to a central server which answers each request: when querying webserver a and b for the same content, the returned binary has at timestamp t_1 the same checksum c_1. At timestamp t_2 — at least one minute later — the servers a and b return both a binary with checksum c_2.

5.4.2 System-Level Behavior

Storm Worm itself is a sophisticated malware binary and uses several advanced techniques, e.g., the binary packer is one of the most advanced seen in the wild [Fra07], the malware uses a rootkit in order to hide its presence on the infected machine, and it has a kernel-level component in order to remain undetected on the system. We do not provide a complete overview of the system-level behavior since some of this information is already available [PSY07, Ste07] and major antivirus vendors periodically publish analysis results of reverse engineering this bot.

We only mention two aspects that are important to understand the network-level behavior of this bot, which is a key part in understanding how to infiltrate and mitigate

this malicious remote control network. First, during the installation process, the malware also stores a configuration file on the infected system. This file contains in an encoded form information about other peers with which the program communicates after the installation phase. Each peer is identified via a hash value and an IP address/port combination. This is the basic information needed to join the peer-to-peer network, for which we provide details in the next section. Based on the CWSandbox analysis report, we can automatically extract this information from a given binary during the behavior-based analysis phase. Second, Storm synchronizes the system time of an infected machine with the help of the Network Time Protocol (NTP). This means that each infected machine has an accurate clock. In the next section, we show how this synchronization is used by Storm for communication purposes.

5.4.3 Network-Level Behavior

For finding other bots within the peer-to-peer network and receiving commands from its controller, the first version of Storm Worm uses OVERNET, a Kademlia-based [MM02] peer-to-peer distributed hash table (DHT) routing protocol. OVERNET is implemented by Edonkey2000, that was officially shut down in early 2006, but still benign peers are online in this network, i.e., not *all* peers within OVERNET are bots per se.

In October 2007, the Storm botnet changed the communication protocol slightly. From then on, Storm does not only use OVERNET for communication, but newer versions use their own peer-to-peer network, which we choose to call the *Stormnet*. This peer-to-peer network is identical to OVERNET except for the fact that each message is XOR encrypted with a 40 byte long key. Therefore, the message types enumerated below remain the same, only the encoding changed. All algorithms introduced in later sections and the general methodology are not affected by this change in communication since the underlying weakness – the use of unauthenticated content-based publish/subscribe style communication – is still present. Note that in Stormnet we do not need to distinguish between bots and benign peers, since only bots participate in this network.

In the following, we describe the network-level communication of Storm and how it uses OVERNET to find other infected peers. As in other DHTs, each OVERNET or Stormnet node has a global identifier, referred to as DHT ID, which is a randomly generated 128 bit ID. When the client application starts for the first time, it generates the DHT ID and stores it. Storm Worm implements the same mechanism and also generates an identifier upon the first startup, which is then used in subsequent communications.

Routing Lookup. Routing in OVERNET and Stormnet is based on prefix matching: a node a forwards a query destined to a node d to the node in its routing table that has the smallest XOR-distance with d. The XOR-distance $d(a,b)$ between nodes a and b is $d(a,b) = a \oplus b$. It is calculated bitwise on the DHT IDs of the two nodes, e.g., the distance between $a = 1011$ and $b = 0111$ is $d(a,b) = 1011 \oplus 0111 = 1100$. The entries in the routing tables are called *contacts* and are organized as an unbalanced *routing tree*. Each contact consists of the node's DHT ID, IP address, and UDP port. A peer a stores only a few contacts to peers that are far away in the DHT ID space (on the left side of

the tree) and increasingly more contacts to peers closer in the DHT ID space (on the right side of the tree).

The left side contains contacts that have no common prefix with the node *a* that owns the routing tree (XOR on the first bit returns 1). The right side contains contacts that have at least one prefix bit in common. The root of the tree is the node *a* itself. The tree is highly unbalanced and the right side of each tree node is (recursively) further divided in two parts containing on the left side the contacts having no further prefix bit in common, and on the right side the contacts having at least one more prefix bit in common. A *bucket* of contacts is attached to each leaf of the routing tree, containing up to ten contacts, which allows to cope with peer churn without the need to periodically check if the contacts are still online. In summary, a node *a* stores only a few contacts that are far away in the overlay and increasingly more peers closer to *a*.

Routing to a given DHT ID is done in an *iterative way*. *P* sends `route request` messages to three peers (to improve robustness against node churn), which may or may not return to *P* `route responses` messages containing new peers even closer to the DHT ID, which are queried by *P* in the next step. The routing lookup terminates when the returned peers are further away from the DHT ID than the peer returning them. While iterative routing experiences a slightly higher delay than recursive routing, it offers increased robustness against message loss and it greatly simplifies crawling the OVERNET network and Stormnet, which we use to enumerate all Storm nodes within these networks (see Section 5.5). In OVERNET and Stormnet, a routing lookup is performed in a first step by both the publish and the search module, which are both used for command distribution by the Storm Worm botnet.

Publishing and Searching. A *key* in a peer-to-peer system is an identifier used to retrieve information. In many peer-to-peer systems, a key is typically published on a single peer that is closest to that key according to the XOR metric. In OVERNET, to deal with node churn, a key is published on twenty different peers. Note that the key is not necessarily published on the peers *closest* to the key. To assure persistence of the information stored, the owner periodically republishes the information.

As for the publishing process, the search procedure uses the routing lookup to find the peer(s) closest to the key searched for. The four most important message types for the publish and search process are:

1. `hello`, to check if the other peer is still alive and to inform the other peer about one's existence and the IP address and DHT ID.

2. `route request/response(kid)`, to find peers that are closer to the DHT ID `kid`.

3. `publish request/response`, to publish information within the DHT.

4. `search request/response(key)`, to search within the DHT for information whose hash is `key`.

The basic idea of the Storm communication is that an infected machine searches for specific keys within the network. The controller knows in advance which keys are searched for by the infected machines and thus she publishes commands at these keys. These keys can be seen as *rendezvous points* or *mailboxes* the controller and infected machines agree on. In the following, we describe this mechanism in more detail.

Storm Worm Communication. In order to find other Storm-infected machines within the OVERNET network, the bot searches for specific keys using the procedure outlined above. This step is necessary since the bot needs to distinguish between regular and infected peers within the network. The key is generated by a function $f(d, r)$ that takes as input the current day d and a random number r between 0 and 31, thus there can be 32 different keys each day. We found this information in two different ways: first, we reverse engineered the bot binary and identified the function that computes the key. The drawback of this approach is that the attacker can easily change f and then we need to analyze the binary again, thus we are always one step behind and have to react once the attacker changes her tactics. Thus a generic way to find the keys is desirable.

The second way to retrieve this information is by treating the bot as a *black-box* and repeatedly force it to re-connect to the network. This is achieved by executing the bot within a honeynet, i.e., a highly controlled environment. The basic idea is to execute the binary on a normal Windows machine, set up a modified firewall in front of this machine to mitigate risk involved (e.g., prohibit sending of spam mails), and capture all network traffic (see Chapter 2 for more details). Since the bot can hardly identify that it runs within a strictly monitored environment, it behaves as normal, connects to the peer-to-peer network, and then starts to search for keys in order to find other infected peers and the commands from the controller. We monitor the communication and extract from the network stream the key the bot searches for. Once we have captured the search key, we revert the honeypot to a clean state and repeat these steps. Since the bot cannot keep any state, it generates again a key and starts searching for it. By repeating this process over and over again, we are able to enumerate the keys used by Storm Worm in a black-box manner, without actually knowing the function f used by the current binary.

Figure 5.1 shows the keys found during a period of five days in October 2007. We see a clear pattern: on each day, there are 32 unique keys which are generated depending on the time, and for different days there is no overlap in the search keys. This result confirms the results of our reverse engineering approach. The keys are important to actually identify Storm-infected machines and we can also use them for mitigation purposes. Another important implication is that we can pre-compute the search keys in advance: on day d, we can set the system time to $d + n$ and perform our black-box enumeration process as outlined above. As a result, we collect all keys the bot will search on day $d + n$ in advance.

If the attackers change the function that generates the key, e.g., by using other inputs for f, we can still determine which keys are *currently* relevant for the communication within the botnet with the help of our honeypot setup: by analyzing the network

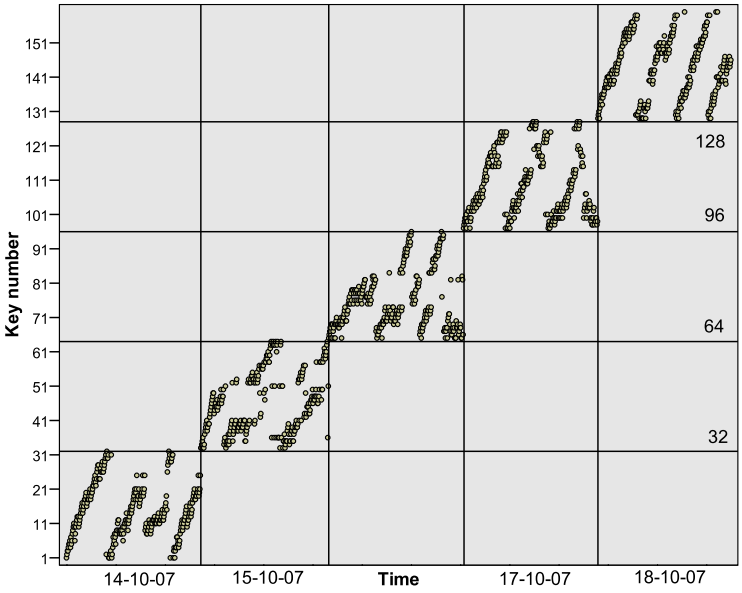


Figure 5.1: Keys generated by Storm in order to find other infected peers within the network (October 14-18, 2007).

communication, we can obtain the current search key relevant for the communication. In general, we can use this setup to learn the keys a bot searches for in a black-box manner, regardless of the actual computation since we impersonate as a valid member of the botnet by executing an actual copy of the bot in the honeypot environment.

The keys are used by the bot to find the commands which should be executed: the attacker has in advance published content at these keys since she knows which keys are searched for by an infected peer. In DHT-based peer-to-peer networks, this is a viable communication mechanism. The actual content published in OVERNET at these keys contains a filename of the pattern "`*.mpg;size=*;`" [PSY07]. No other meta tags (like file size, file type, or codec) are used and the asterisks depict 16-bit numbers. Our observations indicate that the bot computes an IP address and TCP port combination based on these two numbers and then contacts this *control node*. However, up to now we do not know how to compute the IP address and port out of the published numbers.

Only bots participate in Stormnet, thus they do not need to authenticate themselves. Publications in Stormnet do not contain any meta tags. The IP address and port of the machine that send the publish request seem to be the actual information.

All following communication just takes place between the bot and the control node, which sends commands to the bot. This is similar to a two-tier architecture where the first-tier is contained within OVERNET or Stormnet and used to find the second-tier computers that send the actual commands. Once the Storm infected machine has finished the TCP handshake with the control node, this node sends a four byte long challenge c in order to have a weak authentication scheme. The bot knows the secret "key" $k = \texttt{0x3ED9F146}$ and computes the response r via $r = c \oplus k$. This response is

then sent to the control node and the bot is successfully authenticated. All following communication is encoded using *zlib*, a data compression library.

The infected machine receives via this communication channel further commands that it then executes. Up to now, we only observed that infected machines are used to either start DDoS attacks or to send spam e-mails. The DDoS attacks we observed were either SYN or ICMP flooding attacks against various targets all over the Internet. In order to send spam, the infected machines receive a spam template and a list of e-mail addresses to be spammed. We found two different types of mails being sent by Storm: propagation mails that contain different kinds of social engineering campaigns as introduced in Section 5.4.1 or general spam messages that advertise for example pharmaceutical products or stocks. The attackers behind Storm presumably either earn money via renting the botnet to spammers, sending spam on behalf of spammers, or running their own pharmacy shop. Kanich et al. recently analyzed the economic aspects behind Storm Worm in detail [KKL+08a].

5.4.4 Encrypted Communication Within Stormnet

As mentioned above, Stormnet uses encrypted communication and we now describe this procedure in more detail. Several minutes after a bot joins Stormnet, a specific machine X sends a `Publicize` message to the bot. We noticed this behavior for several months (at least between October 2007 and June 2008) and always observed this message coming from the same IP address. After the typical protocol to establish a communication channel within the botnet, we found an interesting pattern: X always sends the same packet which is 180 bytes long (shown in Figure 5.2).

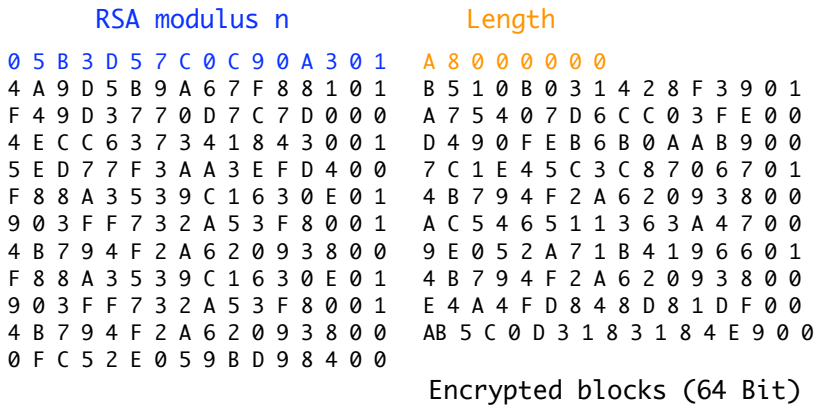


Figure 5.2: Content of RSA-encrypted packets (180 bytes).

Closer examination revealed that the packet content is encrypted with RSA, a public-key encryption system [RSA78]. To encrypt the cleartext p to the ciphertext c, RSA uses the equation

$$c = p^e \bmod n$$

with the public key exponent e and the RSA modulus n. Decryption is implemented in a similar way, using the equation

$$p = c^d \bmod n$$

with the private key exponent d. The first eight bytes within the packet depicted in Figure 5.2 are the RSA modulus n. The next four bytes encode the length of the rest of the packet: $\texttt{0x000000A8} = 168_{10}$ bytes are following in the packet. The private key $d = \texttt{0xBD1AEF19162D5F02}$ is embedded in the bot and can be extracted with the help of reverse engineering the actual binary. Based on this knowledge, we can decrypt the rest of the packet in blocks of 64 bit length. Note that we need to reverse the order of the bytes. For example, the first block can be decrypted in the following way:

$$\begin{aligned} p &= c^d \bmod n \\ &= \texttt{0x0181F8679A5B9D4A}^{\texttt{0x025F2D1619EF1ABD}} \bmod \texttt{0x01A3900C7CD5B305} \\ &= \texttt{0xBF83004E} \end{aligned}$$

The decrypted packet is shown in Figure 5.3. The first two bytes are a checksum of the decrypted content, starting with the second block. Bytes 3 and 4 are the length of the remaining bytes in the packet ($\texttt{0x004E} = 78_{10}$ bytes). 13 blocks with a size of six bytes follow, each of them specifying a TCP port and IP address combination encoded in hexadecimal notation. The TCP port is always set to $\texttt{0x0050} = 80_{10}$, the port typically used for web servers. The IP addresses belong to different co-location providers within the United States. For example, the first two blocks decode to the IP addresses 205.209.179.3 and 69.50.166.234.

```
Checksum  Length     TCP port / IP Address
B F 8 3 0 0 4 E          0 0 5 0 C D D 1 B 3 0 3   0 0 5 0 4 5 3 2 A 6 E A
0 0 5 0 D 8 F F B D D 2  0 0 5 0 D 8 F F B E 1 A   0 0 5 0 4 2 9 4 4 A 0 7
0 0 5 0 4 5 2 9 A B 6 2  0 0 5 0 4 5 2 9 A 2 4 5   0 0 5 0 4 5 2 9 A 2 4 D
0 0 5 0 4 5 2 9 A 1 4 A  0 0 5 0 4 5 2 9 A B 6 2   0 0 5 0 4 5 2 9 A 2 4 5
0 0 5 0 4 5 2 9 A 1 4 A  0 0 5 0 4 5 2 9 B 9 4 2
```

Figure 5.3: Decrypted packets contents.

These specific IP addresses are static, central *control nodes* within Stormnet. Thus the Storm Worm botnet is — contrary to popular belief — no pure peer-to-peer network, but the botnet maintains several central servers as we explain in the following section.

5.4.5 Central Servers Within Stormnet

The Storm Worm botnets does not solely rely on peer-to-peer based communication within the network. In fact, there are several static control nodes acting as a backend within the botnet. We discovered this by dissecting the encrypted packets as explained in the previous section and running actual Storm binaries in different configurations.

Figure 5.4 provides a schematic overview of the network configuration. In total, there are three different layers within the network. First, there are the machines which have a

private IP address, e.g., all machines which are located behind a NAT gateway. These machines are used by the botmasters to send out either spam mails or carry out DDoS attacks. Each of these bots maintains several connections to *gateway bots*. These are infected machines with a public, routable IP address. These machines do not send out spam mails, but are used for relaying messages within the botnet. The third layer consists of the static *control nodes* which for example host spam templates. Furthermore, the infected machines send stolen information from the compromised machines like for example e-mail addresses found on these systems to the control nodes.

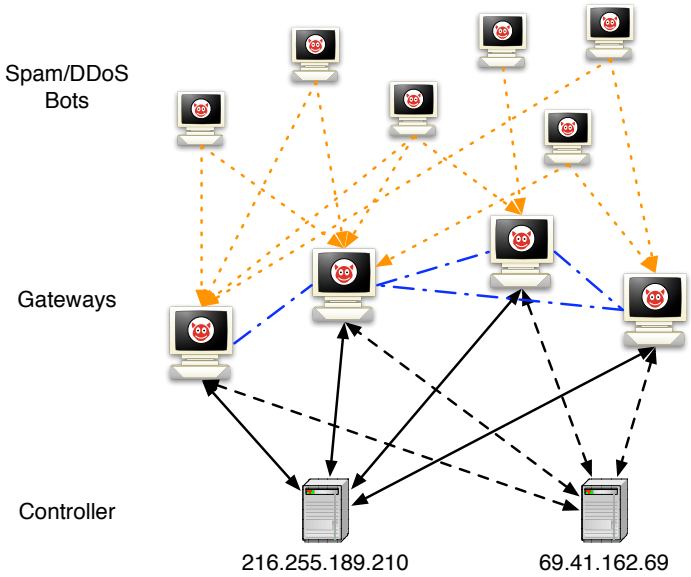


Figure 5.4: Schematic overview of Stormnet, including central servers used for command distribution.

5.5 Tracking of Storm Worm Botnet

After an overview of the behavior of Storm Worm, we now present a case study of how to apply the botnet tracking methodology outlined in Chapter 3 and Section 5.3.2 for this particular bot. We show that we can successfully infiltrate and analyze the botnet, even though there is no central server like in traditional botnets as described in Chapter 4. Furthermore, we also outline possible attacks to mitigate Storm and present our measurement results.

5.5.1 Exploiting the Peer-to-Peer Bootstrapping Process

At the beginning, we need to capture a sample of the bot. As outlined in Section 5.4.1, we can use spamtraps to collect spam mails and then honeyclients to visit the URLs and

obtain a binary copy of the malware. Based on this copy of Storm Worm, we can obtain the current peer list used by the binary via an automated analysis (see Section 5.4.2).

In the first step, we also use the honeynet setup introduced in Section 5.4.3. With the help of the black-box analysis, we are able to observe the keys that Storm Worm searches for. As explained before, the controller cannot send commands directly to the bot, thus the bot needs to search for commands and we exploit this property of Storm to obtain the search keys. During this step we thus obtain (at least a subset of) the current search keys, which allows us to infiltrate and analyze the Storm botnet. With a single honeypot, we were able to reliably acquire all 32 search keys each day for a given Storm Worm binary in a black box manner.

5.5.2 Infiltration and Analysis

Based on the obtained keys and knowledge of the communication protocol used by Storm, we can start with the infiltration and analysis step to learn more about the botnet, e.g., we can enumerate the size of the network. First, we introduce our method to learn more about the peers in OVERNET and Stormnet and about the content announced and searched for in these networks. Afterwards we present several measurement results obtained when tracking this botnet for a longer period of time.

Crawling the Peer-to-Peer Network

To measure the number of peers within the whole peer-to-peer network, we use the crawler for OVERNET and *Stormnet* developed by Steiner as part of his PhD thesis [Ste08]. It uses the same basic principle as the KAD crawler also developed by Steiner [SENB07]. The crawler runs on a single machine and uses a breadth first search issuing `route requests` messages to find peers currently participating in OVERNET or Stormnet. The speed of the crawler allows us to discover all peers within a short amount of time, usually between 20 to 40 seconds (depending on the time of day).

The crawler runs two asynchronous threads: one to send the `route requests` (see Algorithm 1) and one to receive and parse the `route responses` (see Algorithm 2). One list containing the peers discovered so far is maintained and used by both threads. The receiving thread adds the peers extracted from the `route responses` to the list, whereas the sending thread iterates over the list and sends 16 `route requests` to every peer. The DHT ID asked for in the `route requests` are calculated in such a way that each of them falls in different zones of the peer's routing tree. This is done in order to minimize the overlap between the sets of peers returned.

Spying in OVERNET and Stormnet

The main idea of the *Sybil* attack [Dou02] is to introduce malicious peers, the *sybils*, which are all controlled by one entity. Positioned in a strategic way, the *sybils* allow us to gain control over a fraction of the peer-to-peer network or even over the whole network. The *sybils* can monitor the traffic, i.e., act as spies (behavior of the other peers)

Algorithm 1: Send thread (is executed once per crawl)

```
Data: peer: struct{IP address, port number, DHT ID}
Data: shared list Peers = list of peer elements
/* the list of peers filled by the receive thread and worked
   on by the send thread                                      */
Data: int position = 0
/* the position in the list up to which the peers have
   already been queried                                       */
Data: list ids = list of 16 properly chosen DHT ID elements
1 Peers.add(seed); /* initialize the list with the seed peer */
2 while position < size(Peers) do
3     for i=1 to 16 do
4         dest DHT ID = Peers[position].DHT ID ⊕ ids[i];
          /* normalize bucket to peer's position            */
5         send route requests(dest DHT ID) to Peers[position];
6     position++;
```

Algorithm 2: Receive thread (waits for the `route response` messages)

```
Data: message mess = route response message
Data: peer: struct{IP address, port number, DHT ID}
Data: shared list Peers = list of peer elements
/* the list shared with the send thread                       */
1 while true do
2     wait for (mess = route response) message;
3     foreach peer ∈ mess do
4         if peer ∉ Peers then
5             Peers.add(peer);
```

or abuse the protocol in other ways. For example, `route requests` may be forwarded to the wrong end-hosts or re-routed to other *sybil* peers. We use the Sybil attack to infiltrate OVERNET and the Stormnet and observe the communication to get a better understanding of it.

Assume that we want to find out what type of content is published and searched for in the one of both networks in the least intrusive way. For this, we need to introduce *sybils* and make them known, such that their presence is reflected in the routing tables of the non-sybil peers. Steiner developed a light-weight implementation of such a "spy" that is able to create thousands of *sybils* on one single physical machine that we use for our measurements. The spy achieves this scalability since the *sybils* do not keep any state about the interactions with the non-sybil peers [SEENB07]. We introduce 2^{24} *sybils* into OVERNET and Stormnet: the first 24 bits are different for each *sybil* and the following bits are fixed, they are the *signature* of our *sybils*. The spy is implemented in the following steps:

1. Crawl the DHT ID space using our crawler to learn about the set of peers $\mathscr{P}$ currently online.

2. Send `hello requests` to the peers $\mathscr{P}$ in order to "poison" their routing tables with entries that point to our *sybils*. The peers that receive a `hello request` will add the *sybil* to their routing table.

3. When a `route request` initiated by non-sybil peer P reaches a *sybil*, that request will be answered with a set of *sybils* whose DHT IDs are closer to the target. This way, P has the impression of approaching the target. Once P is "close enough" to the target DHT ID, it will initiate a `publish request` or `search request` also destined to one of our *sybil* peers. Therefore, for any `route request` that reaches one of our *sybil* peers, we can be sure that the follow-up `publish request` or `search request` will also end-up on the same *sybil*.

4. Store the content of all the requests received in a database for later evaluation.

Using the Sybil attack, we can monitor requests within the whole network.

5.6 Empirical Measurements on Storm Botnet

Other Studies related to Storm Worm. Concurrent to our work, Storm Worm has become the subject of intense studies at many places around the world [Enr07, Bal07]. By looking at the (DHT ID, IP address) pairs collected by our crawler, we found several instances where DHT IDs that contain a well chosen pattern covered the whole DHT ID space and the IP addresses map all to the same institution. We could observe experiments (or worm activities) going on in San Diego (UCSD), Atlanta (Georgia Tech) and many other places (also on address spaces we could not resolve). We filtered out all these (DHT ID, IP address) pairs before doing our analysis.

In their technical report, Sarat and Terzis [ST07] claim that OVERNET is used by 430,000 to 600,000 peers. This contradicts our results and the results found in the literature. Moreover, they claim that all these nodes are infected with Storm and they do not differentiate between bots and regular peers. We also found many legitimate peers within OVERNET and thus they over-estimate the size of the botnet created by Storm. They observed several IP addresses that account for almost half of all DHT IDs. Since their observation period is the same than ours, we believe that they saw our *sybils* and those of other researchers working on the Storm Worm botnet.

5.6.1 Size Estimations for Storm Bots in OVERNET

During full crawls from October 2007 until the beginning of February 2008, we found between 45,000 and 80,000 concurrent online peers in OVERNET. We define a peer as the combination of an IP address, a port number and a DHT ID. In the remaining part, we use the term "IP address" for simplicity. If one DHT ID is used on several IP addresses, we filter these instances out. We also filter out IP addresses that run more than one DHT ID simultaneously.

Note that the same machine participating in OVERNET can, on the one hand, change its IP address over time. This fact is known as *IP address aliasing*. On the other hand, it can also change its DHT ID over time. This is known as DHT ID *aliasing*. Due to this reason, simply counting the number of different DHT IDs or IP addresses provides only a rough estimate of the total number of machines participating in OVERNET. Nevertheless, we present for the sake of completeness the total number of DHT IDs and IP addresses observed during the month of October 2007: with our crawler, we could observe 426,511 different DHT IDs on 1,777,886 different IP addresses. These numbers are an upper bound for the number of Storm-infected machines: since we enumerate the whole DHT ID space, we find all online peers, from which a subset is infected with a copy of the Storm Worm bot.

About 75% of all peers are not located behind NATs or firewalls and can be *directly contacted* by our crawler. We used the MaxMind database [Max] to map the IP addresses to countries. We saw clients from 210 countries. These split up in 19.8% that cannot be resolved, 12.4% from the US, 9.4% from Uruguay, 6% from Germany, etc.

Lower Bound for Storm-infected Machines in OVERNET. When spying on OVERNET, the benign peers can be distinguished from the bots of the Storm Worm botnet: bots publish files with characteristic filenames and no other meta tags (see Section 5.4.3 for details). However, not every bot does make such announcements. This allows us to obtain a lower bound of the size of the botnet since only peers with this characteristic pattern are definitely infected with Storm. Note that only the Storm bots with a public IP address publish content in OVERNET.

Every day we see between 5,000 and 6,000 distinct peers that publish Storm related content. About the same number of peers publish real, e.g., non-Storm related, content. Around 30,000 peers per day did perform searches for content. Most of the clients that published Storm content (the bots running on public IP addresses) come from the US

(31%) followed by India (5.3%) and Russia (5.2%). Note, however, that 21% of the IP addresses could not be mapped to any country. In total, we observed bots from over 133 countries. Due to the fact that all social engineering campaigns we observed contain English text, it is not surprising that the majority of Storm-infected machines are located in the United States.

Estimating the Number of Storm-infected Machines in OVERNET. All we can measure with confidence are upper and lower bounds of the number of concurrently active bots in OVERNET. The lower bound being around 5,000 – 6,000 and the upper bound being around 45,000 – 80,000 distinct bots.

Storm content was published using roughly 1,000 different keys per day. This indicates that there are many different version of Storm in the wild, since each binary only searches for 32 keys per day. Some of these keys were used in more than 1,500 publications requests, whereas the majority of keys was used in only few publications. During the observation period in October 2007, we observed a total of 13,307 keyword hashes and 179,451 different file hashes. We found that 750,451 non-Storm related files were announced on 139,587 different keywords.

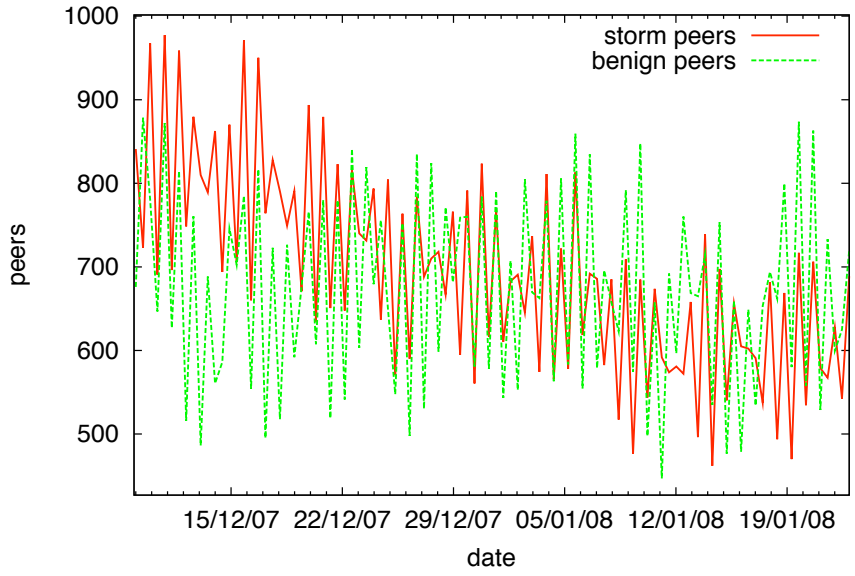


Figure 5.5: Number of bots and benign peers that published content in OVERNET.

From the end of the year 2007 on, the number of storm bots using OVERNET and the Storm activity in OVERNET decreased. Figure 5.5 shows the number of bots and benign peers that published content in OVERNET in December 2007 and January 2008. The number of benign peers remains constant, while the number of storm bots decreases. We think this is due to the fact that the whole botnet shifted from OVERNET to Stormnet.

5.6.2 Size Estimation for Stormnet

We can apply the algorithms outlined above to enumerate all peers within Stormnet. The important difference between Stormnet and OVERNET is the fact that OVERNET is used by regular clients and Storm bots, whereas Stormnet is used only by machines infected with Storm Worm. Hence, we do not need to differentiate between benign and infected peers within Stormnet.

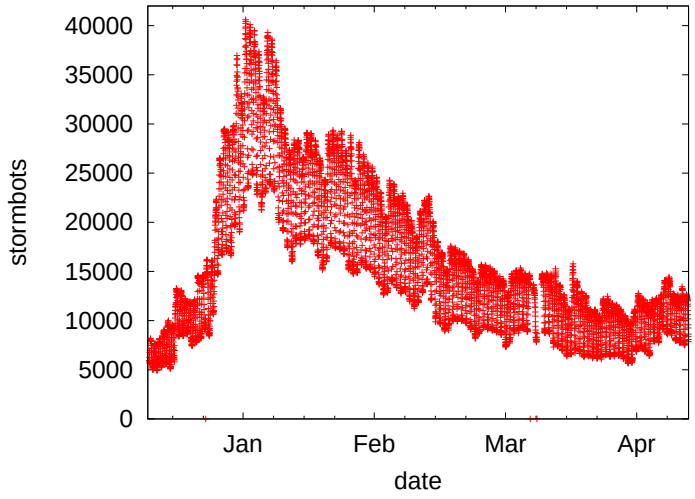


Figure 5.6: Total number of bots in Stormnet.

In collaboration with Steiner, we crawled Stormnet every 30 minutes since beginning of December 2007 until the middle of June 2008. During this period, we saw between 5,000 and 40,000 peers concurrently online. There was a sharp increase in the number of storm bots at the end of 2007 due to a propagation wave during Christmas and New Years Eve (Figure 5.6). After that increase, the number of bots varied between 25,000 and 40,000 before stabilizing in the beginning of March 2008 around 10,000 to 15,000 bots. In total, we found bots in more than 200 different countries. The biggest fraction comes from the US (23%). As seen in the Figure 5.7, Storm Worm also exhibits strong diurnal patterns like other botnets [DZL06].

Figure 5.8 depicts the number of distinct IP addresses as well as the number of distinct "rendezvous" hashes searched for in Stormnet. At the end of 2007, the number of peers searching in Stormnet increased significantly and the search activity stabilized at high level in the middle of January 2008, before increasing significantly since February 2008. Similar to the diurnal pattern of the number of storm bots, also the search activity within Stormnet shows a distinct diurnal pattern.

The publish activity shows exactly the same behavior over time compared to the search activity (Figure 5.9). However, the number of "rendezvous" hashes that are searched for is nearly of an order of magnitude higher than the number of hashes that are published. For the number of distinct IP addresses, especially in the week from

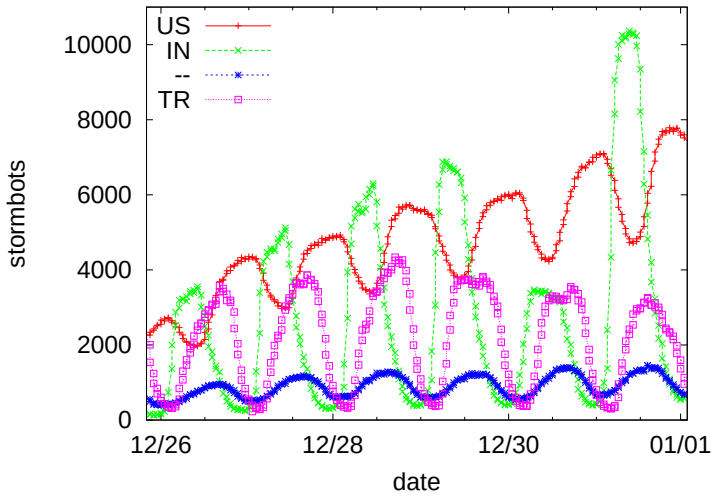


Figure 5.7: Detailed overview of number of bots within Stormnet, split by geo-location.

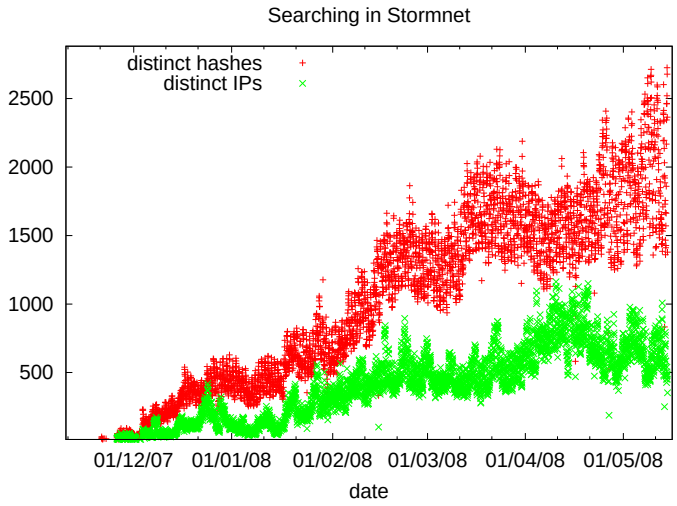


Figure 5.8: Search activity in Stormnet.

29/12/2007 to 05/01/2008 and starting again on 19/01/2008, the distinct number of IP addresses launching search queries are two orders of magnitudes higher than the number of IP addresses publishing. The number of IP addresses searching for content is around three times bigger than the number of IP addresses publishing content. It is somehow intuitive that the number of IP addresses that search is bigger than those publishing, since the goal is to propagate information.

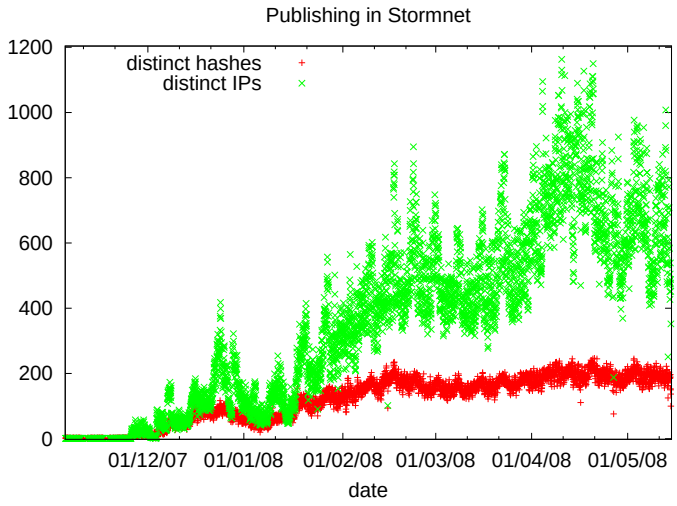


Figure 5.9: Publish activity (distinct IP addresses and rendezvous hashes) in Stormnet.

5.7 Mitigation of Storm Worm Botnet

Based on the information collected during the infiltration and analysis phase, we can also try to actually mitigate the botnet. In this section, we present two theoretical approaches that could be used to mitigate Storm Worm and our empirical measurement results for both of them.

5.7.1 Eclipsing Content

A special form of the *sybil* attack is the *eclipse* attack [SNDW06] that aims to separate a part of the peer-to-peer network from the rest. The way we perform an eclipse attack resembles very much that of the *sybil* attack described above, except that the DHT ID space covered is much smaller.

To eclipse a particular keyword K, we position a certain number of *sybils* closely around K, i.e., the DHT IDs of the *sybils* are closer to the hash value of K than the DHT IDs of any real peer. Afterwards, we need to announce these *sybils* to the regular peers in order to "poison" the regular peers' routing tables and to attract all the `route requests` for keyword K. Unfortunately, using this technique we could — in contrast to similar experiments in KAD [SBEN07] — not completely eclipse a particular keyword. This is due to the fact that in OVERNET and Stormnet the content is spread through the entire hash space and not restricted to a zone around the keyword K. As a consequence, in OVERNET and Stormnet, the eclipse attack cannot be used to mitigate the Storm Worm network.

5.7.2 Polluting

Since eclipsing content is not feasible in OVERNET or Stormnet, we investigated another way to control particular content. To prevent peers from retrieving search results for a certain key K, we publish a very large number of files using K. The goal of the pollution attack is to "overwrite" the content previously published under key K. Since the Storm bots continue to publish their content as well, this is a race between the group performing mitigation attempts and the infected machines.

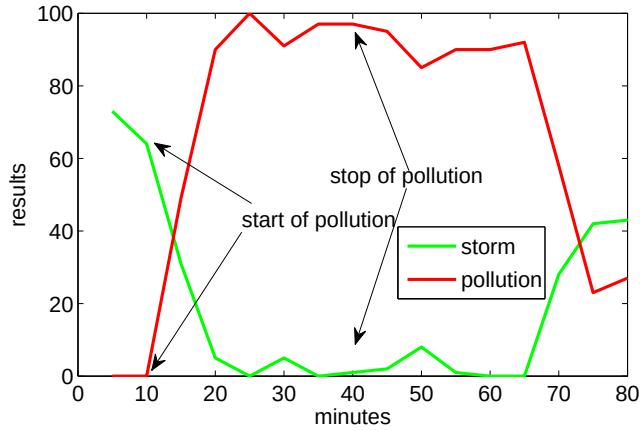
To perform this attack, we again first crawl the network, and then publish files to all those peers having at least the first 4 bits in common with K. This crawling and publishing is repeated during the entire attack. A publishing round takes about 5 seconds, during which we try to publish on about 2,200 peers, out of which about 400 accept our publications. The peers that do not respond did either previously leave the network, could not be contacted because they are behind a NAT gateway, or are overloaded and could not process our publication.

Once a search is launched by any regular client or bot, it searches on peers closely around K and will then receive so many results (our fake announcements) that it is going to stop the search very soon and not going to continue the search farther away from K. That way, publications of K that are stored on peers far away from K do not affect the effectiveness of the attack as they do for the eclipse attack.

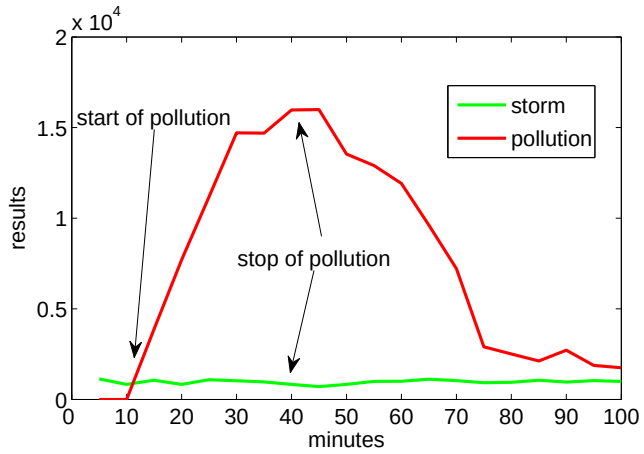
We evaluate the effectiveness of the pollution attack by polluting a hash used by Storm and searching at the same time for that hash. We use two different machines, located at two different networks. For searching we use `kadc` [Kad], an open-source OVERNET implementation, and an exhaustive search algorithm developed by Steiner. Our search method is very intrusive, it crawls the entire network and asks every peer for the specified content with key K. Figure 5.10a shows that the number of Storm content quickly decreases in the results obtained by the regular search algorithm, then nearly completely disappears from the results some minutes after the attack is launched, and finally comes back after the attack is stopped. However, by modifying the search algorithm used and by asking all peers in the network for the content and not only the peers close to the content's hash, the storm related content can still be found (Figure 5.10b). Our experiments show that by polluting all those hashes that we identified to be *storm hashes*, we can disrupt the communication of the botnet. The bots still publish and search, but they do not receive the intended results from the control nodes and can thus not carry out commands by the botmaster.

5.8 Summary

In this chapter, we showed how to use the methodology of botnet tracking for botnets which use peer-to-peer techniques for communication. We exemplified our methodology with a case study on Storm Worm, the most wide-spread peer-to-peer bot propagating in the wild at the time of writing. Our case study focussed on the communication within the botnet and especially the way the attacker and the bots communicate with each other. Storm Worm uses a two-tier architecture where the first-tier is contained within



(a) Using the standard search.



(b) Using the exhaustive search.

Figure 5.10: The number of publications by Storm bots vs. the number of publications by our pollution attack.

the peer-to-peer networks OVERNET and the Stormnet and used to find the second-tier computers that send the actual commands. We could distinguish the bots from the benign peers in the OVERNET network and identify the bots in the Stormnet and give some precise estimates about their numbers. Moreover, we presented two techniques to disrupt the communication of the bots in both networks. While eclipsing is not very successful, polluting proved to be very effective.

Chapter 6

Tracking Fast-Flux Service Networks

6.1 Introduction

Up to now, the focus of this work was on "classical" botnets, i.e., networks of compromised machines that directly execute the attacker's commands such as performing DDoS attacks. In such malicious remote control networks, the compromised machines either contact a central server (see Chapter 4) or use a communication channel based on peer-to-peer techniques (see Chapter 5) to receive commands from the attacker. In this chapter, we focus on a different kind of malicious remote control networks, namely *fast-flux service networks* (FFSNs). The basic idea of such networks is that the attacker establishes a distributed proxy network on top of compromised machines that redirects traffic through these proxies to a central site, which hosts the actual content the attacker wants to publish. Instead of using the infected machines to send out spam mails, perform DDoS attacks, or mass identity theft, the focus of FFSNs thus lies in constructing a robust hosting infrastructure with the help of the victim's machines.

In this chapter, we show how the basic methodology proposed in Chapter 3 can also be applied to FFSNs. In the first phase, we need to get hold of a sample of a *fast-flux agent*, i.e., a sample belonging to a FFSN. Since this kind of malware commonly propagates further by drive-by downloads on malicious websites, we can use client-side honeypots to collect samples of such agents. By executing a fast-flux agent in a honeypot environment, we can then closely study the communication channel used by a FFSN. Since FFSNs are based on DNS, we can also *passively* infiltrate the network, i.e., by repeatedly performing DNS lookups for domains used by FFSNs, we can also get an insight into their operation. All collected information can then in the final phase be used to mitigate the threat and prevent further abuses.

While the technical aspects of fast-flux service networks have been reported on elsewhere [The07], it is still unclear how large the threat of FFSNs is and how these networks can be mitigated. In this chapter we present one of the first empirical studies of the fast-flux phenomenon, giving many details about FFSNs we observed during a two-month period in summer 2007. By analyzing the general principles of these

networks, we develop a metric with which FFSNs can be effectively detected using information offered by DNS. Such a metric can be used to develop methods to mitigate FFSNs, a topic which we also discuss in this chapter.

Contributions. The contributions of this chapter are threefold:

- We show that the methodology proposed in Chapter 3 can also be applied to FFSNs, a completely different type of malicious remote control networks compared to "classical" botnets.
- We present the first detailed empirical study of FFSNs and measure with the help of several properties the extent of this threat. Our measurements show that almost 30% of all domains advertised in spam are hosted via FFSNs and we found several ten thousands of compromised machines during our study.
- We analyze the principles of FFSNs, from this develop a metric to detect FFSN and show the effectiveness of this metric by using real-world measurements.

Our contribution does not only clarify the background behind a fast growing threat within the Internet, but moreover we presents new results obtained after reinterpreting previous measurements [AFSV07] taking our findings into consideration. We are aware neither of any other work which has investigated FFSNs in an empirical way nor of work which discusses detection and mitigation strategies. Due to the severity of the novel FFSN threat, we feel that our results can also be helpful in practice.

Outline. Section 6.2 motivates our work and in Section 6.3, we provide an overview of the technical background related to this chapter. We develop a metrical framework to automatically detect FFSNs based on their answers to DNS requests in Section 6.4 and show that our metric can identify FFSNs with high accuracy and almost no false positives. We present a method to track FFSNs according to our methodology in Section 6.5 and in Section 6.6, we provide detailed results of empirical measures for several FFSNs. Several strategies to mitigate FFSNs are proposed in Section 6.7 before we conclude this chapter in Section 6.8 with a summary.

6.2 Motivation

One of the most important properties of commercial service providers on the Internet is the continuous *availability* of their websites. If webservers are not online, the service cannot be offered, resulting in loss of profit. It is estimated that online shops like Amazon loose about $550,000 for every hour that their website is not online [Pat02]. The main cause of unavailability have been *hardware faults*: since electronic components have only a limited lifetime, computer and storage systems are prone to failures. Techniques from the area of *reliability engineering* [Bir04], e.g., redundancy techniques like RAID [PGK88] or commercial failover systems [Hew07], help to overcome these failures.

Nowadays, tolerance against the failure of individual hardware components is rather well-understood. However, *Distributed Denial-of-Service attacks* [MR04], especially in the form of network bandwidth exhaustion, pose a significant threat. This threat has become an almost daily nuisance with the advent of *botnets*, which we introduced in detail in the previous two chapters. A medium-sized botnet of 1,000 or 2,000 machines is often sufficient to take down almost any network service.

Several methods exist to alleviate the results of distributed denial-of-service attacks. We focus here on standard methods which use the global *Domain Name Service* (DNS). A well-known method is called *Round-robin DNS* (RRDNS) [Bri95]. This method is used by large websites in order to distribute the load of incoming requests to several servers [KBM94, CCY99] at a single physical location. A more advanced (and more expensive) technique is implemented by so called *Content Distribution Networks* (CDNs) like *Akamai* [Aka]. The basic idea is to distribute the load not only to multiple servers at a single location, but to also distribute these servers over the globe. The real benefit of CDNs comes with using DNS: when accessing the name of the service via DNS, the CDN computes with the help of sophisticated techniques the "nearest" server (in terms of network topology and current link characteristics) and returns its IP address. The client then establishes a connection to this server and retrieves the content from there. In effect, content is thereby moved "closer" to the client that sends the request, increasing responsiveness and availability.

RRDNS and CDNs are techniques employed by *legal* commercial organizations. Unfortunately, there are also a lot of *illegal* commercial organizations offering services on the Internet. Naturally, they also demand high availability for the services they offer. For example, a *spammer* that runs a website to sell pharmaceutical products, adult content, or replica watches can only make money if the website is reachable. As another example, consider a *phisher* that steals confidential information by redirecting users to fake online sites and tricking them into revealing their credentials. The phisher also requires the phishing website to be online most of the time; only then victims can fall for this scam. As a final example, consider a *botherder* who directs a large botnet. The botnet itself requires a reliable hosting infrastructure such that the botherder's commands can be sent to the bots or malicious binaries can be downloaded by existing bots.

The starting point of our research is the question, how illegal organizations achieve high availability of their online services, explicitly focusing on HTTP services (i.e., websites). The problems which such organizations face today is that law enforcement's abilities to take webservers with illegal content down has reached a certain level of effectiveness. RRDNS and CDNs are therefore no real alternatives for hosting scams. In a slightly ironic repetition of history, it seems that today illegal organizations are discovering classic redundancy techniques to increase the resilience of their infrastructure, as we explain in this chapter.

In this chapter we focus on a newly emerging threat on the Internet called a *fast-flux service network* (FFSN). Such a network shows a similar behavior as RRDNS and CDNs in regards to the network characteristics: a single service seems to be hosted by many different IP addresses. Roughly speaking, a FFSN uses rapid-changing DNS entries to build a hosting infrastructure with increased resilience. The key idea is to construct

a distributed proxy network on top of compromised machines that redirects traffic through these proxies to a central site, which hosts the actual content. Taking down any of the proxies does not effect the availability of the central site: with the help of a technique similar to RRDNS, the attacker always returns a different set of IP addresses for a DNS query and thus distributes the traffic over the whole proxy network. This leads to an increased resilience since taking down such schemes usually needs cooperation with a domain name registrar. As we will see in this chapter, a single fast-flux service network can consist of hundreds or even thousands of compromised machines which form a different kind of malicious remote control network. The general methodology we introduced in Chapter 3 can also be used to track this kind of malicious remote control networks as we shown in the rest of this chapter.

6.3 Technical Background

6.3.1 Round-Robin DNS

Round-robin DNS is implemented by responding to DNS requests not with a single DNS *A record* (i.e., hostname to IP address mapping), but a *list* of A records. The DNS server cycles through this list and returns them in a round-robin fashion.

```
;; ANSWER SECTION:
myspace.com.            3600    IN      A       216.178.38.116
myspace.com.            3600    IN      A       216.178.38.121
myspace.com.            3600    IN      A       216.178.38.104
```

Figure 6.1: Example of round-robin DNS as used by `myspace.com`.

Figure 6.1 provides an example of round-robin DNS used by `myspace.com`, one of the Top 10 websites regarding traffic according to Alexa [Ale07]. We performed the DNS lookup with *dig* [Int07], a tool dedicated to this task, and only show the `ANSWER` section for the sake of brevity. In total, three A records are returned for this particular query, all pointing to servers hosting the same content. The DNS client can then choose one of these A records and return the corresponding IP address. Basic DNS clients simply use the first record, but different strategies can exist, e.g., using the record which is closest to the DNS client in terms of network proximity. Every A record also has a *Time To Live* (TTL) for the mapping, specifying the amount of seconds the response remains valid. RFC 1912 recommends minimum TTL values of 1-5 days such that clients can benefit from the effects of DNS caching [Bar96]. Shaikh et al. study the trade-off between responsiveness of round-robin based server selection, client-perceived latency, and overall scalability of the system and show that small TTL values can have negative effects [STA01]. If the DNS lookup is repeated while the answer is still valid, the query for `myspace.com` returns the same set of IP addresses, but in a different order. Even after the TTL has expired, i.e., after 3600 seconds in this example, a subsequent DNS lookup returns the same set of A records.

We tested all domains from the Alexa Top 500 list and found that almost 33 percent use some form of RRDNS, i.e., more than one A record was returned in a DNS lookup. Furthermore, we measured the TTL values used by these sites: about 43 percent of these domains have a TTL below 1800 seconds.

6.3.2 Content Distribution Networks

Like round-robin DNS, content distribution networks also usually implement their service using DNS [GCR01, KWZ01, SGD+02]: The domain name of the entity which wants to host its content via a CDN points to the nameservers of the CDN. With the help of sophisticated techniques, the CDN computes the (in terms of network topology and current link characteristics) nearest *edge server* and returns the IP address of this server to which the client then connects.

```
;; ANSWER SECTION:
images.pcworld.com.                   900  IN  CNAME  images.pcworld.com.edgesuite.net.
images.pcworld.com.edgesuite.net.   21600  IN  CNAME  a1694.g.akamai.net.
a1694.g.akamai.net.                    20  IN      A  212.201.100.135
a1694.g.akamai.net.                    20  IN      A  212.201.100.141
```

Figure 6.2: Example of DNS lookup for domain `images.pcworld.com` hosted via Content Distribution Network, in this case Akamai.

Figure 6.2 depicts the A and CNAME (*canonical name*, an alias for one name to another) records returned in DNS lookups for the domain `images.pcworld.com`, which uses Akamai to host its content. Again, the DNS lookup returns multiple A records which all belong to the network of Akamai. Compared to the previous example, the TTL is significantly lower with only 20. A low TTL is used by CDNs to quickly enable them to react to changes in link characteristics. The Akamai edge server is automatically picked depending on the type of content and the user's network location, i.e., it can change over time for a given end-user.

6.3.3 Fast-Flux Service Networks

From an attacker's perspective, the ideas behind round-robin DNS and content distribution networks have some interesting properties. For example, a spammer is interested in having a high reliability for hosting the domain advertised in his spamming e-mails. If he could advertise several IP addresses for a given domain, it would become harder to shut down the online pharmacy shop belonging to the scam: if at least one of the IP addresses returned in an A record is reachable, the whole scam is working. As another example, a botherder is interested in scalability and he could use round-robin DNS to split the bots across multiple Command & Control servers in order to complicate shutdown attempts. In both examples, the resulting scam infrastructure is more resilient to mitigation attempts.

RRDNS and CDNs return several IP addresses in response to a DNS request. As long as one of these addresses responds, the entire service is online. Fast-flux service networks employ the same idea in an innovative way. The main characteristic of fast-flux is the rapid (*fast*) change in DNS answers: a given fast-flux domain returns a few A records from a larger pool of compromised machines ("flux-agents") and returns a *different* subset after the (low) TTL has expired. By using the compromised machines as proxies to route an incoming request to another system (control node or "mothership"), an attacker can build a resilient, robust, one-hop overlay network on top of the compromised machines.

We explain the structure behind FFSNs with the help of a short example. The domain `unsubscribe-link.com` was found in a spam e-mail in November 2008. The *dig* response for this domain is shown in the upper part of Figure 6.3. We repeated the DNS lookup after the TTL timeout given in the first answer to have two consecutive lookups of the same domain. The results of the second lookup is shown in the lower part of Figure 6.3. The DNS request returns five A records, similar to the round-robin DNS example above. However, all IP addresses belong to different network ranges. We performed a reverse DNS lookup, resolved the Autonomous System Number (ASN), and determined the country via geolocation lookup for each of the IP addresses returned in the first lookup. The results are shown in Table 6.1. Overall, we identify several interesting features: first, all IP addresses are located in DSL/dial-up network ranges located in several different countries, e.g., United States, Italy, and Poland. Second, the IP addresses belong to several different Autonomous Systems (AS). Third, the TTL is rather low with 1800 seconds. And fourth, the DNS server returns a set of five totally different IP addresses in the second request.

```
 ;; ANSWER SECTION:
unsubscribe-link.com.     1800    IN      A       76.19.165.94
unsubscribe-link.com.     1800    IN      A       83.24.202.150
unsubscribe-link.com.     1800    IN      A       87.6.20.246
unsubscribe-link.com.     1800    IN      A       222.147.245.25
unsubscribe-link.com.     1800    IN      A       70.244.130.148
```

```
;; ANSWER SECTION:
unsubscribe-link.com.     1800    IN      A       118.219.111.107
unsubscribe-link.com.     1800    IN      A       88.109.35.223
unsubscribe-link.com.     1800    IN      A       89.201.100.21
unsubscribe-link.com.     1800    IN      A       79.114.237.237
unsubscribe-link.com.     1800    IN      A       82.83.234.227
```

Figure 6.3: Example of A records returned for two consecutive DNS lookups of domain found in spam e-mail. The DNS lookups were performed 1800 seconds apart such that the TTL expired after the first request.

A closer examination reveals that the A records returned by the DNS lookup point to IP addresses of suspected compromised machines which run a so called *flux-agent*. The flux-agents are basically proxies which redirect an incoming request to the *control node* [The07], on which the actual content of the scam is hosted.

IP address	Reverse DNS lookup for IP address	ASN	Country
76.19.165.94	c-76-19-165-94.hsd1.ct.comcast.net.	7015	US
83.24.202.150	drq150.neoplus.adsl.tpnet.pl.	5617	Poland
87.6.20.246	host246-20-dynamic.6-87-r.retail.telecomitalia.it.	3269	Italy
222.147.245.25	p2025-ipbf705osakakita.osaka.ocn.ne.jp.	4713	Japan
70.244.130.148	ppp-70-244-130-148.dsl.lbcktx.swbell.net.	7132	US

Table 6.1: Reverse DNS lookup, Autonomous System Number (ASN), and country for first set of A records returned for fast-flux domain from Figure 6.3.

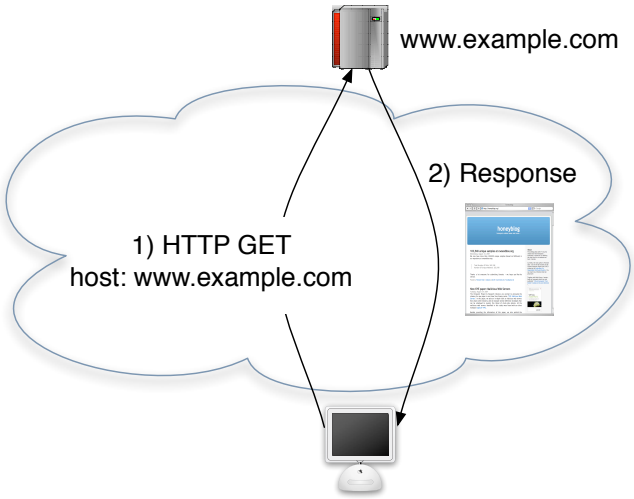


Figure 6.4: Content retrieval process for benign HTTP server.

Figure 6.4 illustrates the process of retrieving the content from a legitimate site: the client contacts the webserver and the content is sent directly from the server to the client. This is a common setup used by many websites.

In a scam that uses FFSN for hosting, the process is slightly different (Figure 6.5): The client uses DNS to resolve the domain and then contacts one of the flux-agents. The agent relays the request to the control node, which sends the content to the flux-agent. In the fourth step, the content is delivered to the client. Note that if the TTL for the fast-flux domain expires and the client performs another DNS lookup, the DNS lookup process will most likely return a different set of A records. This means that the client will then contact another flux-agent, but the request is relayed from that machine to the control node in order to retrieve the actual content. More technical details on fast-flux service networks can be found in a recent paper by the Honeynet Project [The07].

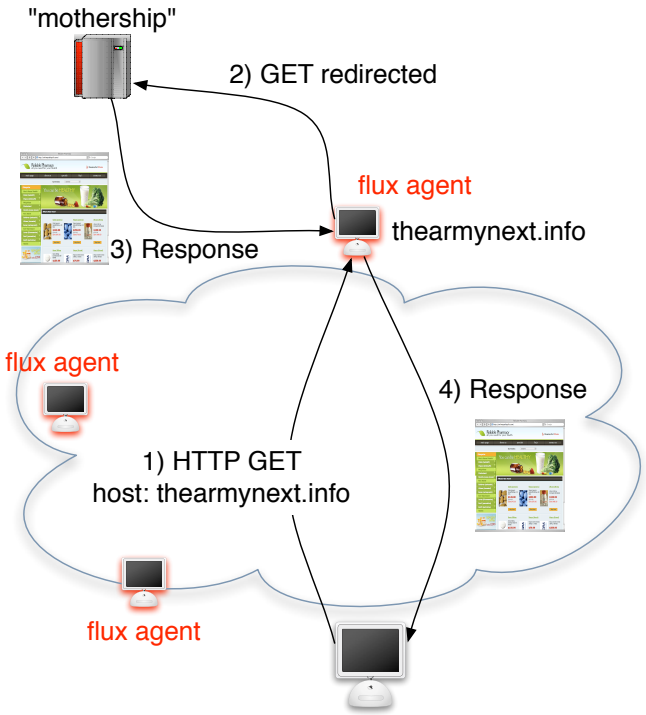


Figure 6.5: Content retrieval process for content being hosted in fast-flux service network.

6.4 Automated Identification of Fast-Flux Domains

As we want to distinguish between FFSNs and other legitimate domains in an automated way, we now turn to the extraction of features enabling us to decide whether a given domain is using the FFSN infrastructure or not.

Restrictions in establishing a FFSN. In contrast to legitimate service providers which may buy availability over CDNs, providers running FFSNs naturally suffer from two main restrictions:

- *IP address diversity*: A scammer is not as free to choose the hardware and network location (IP address) of an individual node as freely as in a CDN. Basically, the FFSN has to live with those machines which can be compromised to run a flux-agent. The range of IP addresses must therefore be necessarily rather diverse and the attacker cannot choose to have a node with a particular IP address.
- *No physical agent control*: In contrast to CDNs which run in large computing centers which professionally host the servers and manage server failures through planned downtimes, a scammer does not have direct control over the machines which run the FFSN. Even worse, flux-agents usually run on ill-administered

machines in dial-up networks which may go down any minute even if their uptime is rather large. This implies that there is no guaranteed uptime of the flux-agent the scammer can rely on.

Possible distinguishing parameters. Based on these two restrictions in establishing a FFSN, we now enumerate a set of parameters which can be used to distinguish DNS network behavior of CDNs from FFSNs. The absence of physical control over the flux-agents results in the consideration of the following two values:

- n_A, the number of unique A records returned in all DNS lookups: legitimate domains commonly return only one to three A records, whereas fast-flux domains often return five or more A records in a single lookup in order to have a higher guarantee that at least one of the IPs is online.

- n_{NS}, the number of nameserver (NS) records in one single lookup: FFSNs can also host the nameserver within the fast-flux network [The07] and often return several NS records and A records for the NS records. In contrast, legitimate domains commonly return a small set of NS records.

The restriction of IP address diversity results in the consideration of the following value:

- n_{ASN}, the number of unique ASNs for all A records: Legitimate domains and even the domains hosted via CDNs tend to return only A records from one particular AS. In contrast, FFSNs tend to be located in different ASs since the infected machines are scattered across different ISPs.

All the above parameters can be determined via DNS lookups and short post-processing of the result. Note that we do not consider the TTL value of the DNS entries as a good parameter. This is because legitimate domains like those hosted via CDNs have similar requirements as FFSNs with respect to the speed of adaptation to network congestion or server outages. The TTL value is, however, a good indicator to distinguish FFSN/CDN from RRDNS. Therefore we take only domains with a TTL of the A records below or equal to 1800 seconds into account, since higher TTL values cannot be considered *fast* enough for rapid changes.

Fluxiness. In general, a metric to distinguish FFSNs from CDNs is a function of n_A, n_{AS}, and n_{NS}. Several possibilities to define this function exist. For example a first approximation could be the following value, which we call the *fluxiness* of a domain:

$$\varphi = n_A / n_{single}$$

The value n_{single} is the number of A records a single lookup returns. A value $\varphi = 1.0$ means that the set of A records remains constant over several consecutive lookups, which is common for benign domains. In contrast, $\varphi > 1.0$ indicates that at least one new A record was observed in consecutive requests, a strong indication of CDNs and

FFSNs. In the example of Figure 6.3, $\varphi = 2.0$ since the second set of returned A records has no overlap with the first lookup.

Note that the fluxiness of a domain is implicitly also contained in the two features n_A and n_{ASN}: for FFSNs (and also CDNs), the number of observed A records (and thus potentially also number of ASNs) grows over time since the lookup process returns a different set of IPs over time.

Flux-Score. A general metric for detection of fast-flux domains can be derived by considering the observed parameters as vectors x of the form (n_A, n_{ASN}, n_{NS}). The resulting vector space enables definition of a linear decision function F using a weight vector w and a bias term b by

$$F(x) = \begin{cases} w^T x - b > 0 & \text{if } x \text{ is a fast-flux domain} \\ w^T x - b \leq 0 & \text{if } x \text{ is a benign domain} \end{cases}$$

The decision surface underlying F is the hyperplane $w^T x + b = 0$ separating instances of fast-flux service networks from benign domains.

Given a corpus of labeled fast-flux and benign domains, there exist numerous assignments of w and b correctly discriminating both classes, but differing in their ability to *generalize* beyond the seen data. A well-known technique for obtaining strong generalization is determining the *optimal hyperplane*, which separates classes with maximum margin [Vap98]. For the linear case of the decision function F, an optimal hyperplane can be efficiently computed using the technique of linear programming [BM00].

Based on a labeled corpus of domains, we can determine a decision function F with high generalization ability by computing the weight vector w and bias b of the optimal hyperplane. The decision function F induces a scoring metric f for the detection of fast-flux domains referred to as *flux-score* and given by

$$f(x) = w^T x = w_1 \cdot n_A + w_2 \cdot n_{ASN} + w_3 \cdot n_{NS} \tag{6.1}$$

A flux-score $f(x) > b$ indicates an instance of a fast-flux service network, while lower scores correspond to benign domains. Furthermore, the flux-score provides a *ranking* of domains, such that higher values reflect a larger degree of fast-flux characteristics — implicitly corresponding to a larger distance from the optimal hyperplane of F.

Validation of Current FFSN. To instantiate the weights w_1, w_2, and w_3, we used empirical measurements of 128 manually verified fast-flux domains and 5,803 benign domains as input. The latter were randomly taken from the Open Directory Project [Net07], a human-edited directory, and the Alexa Top 500 list. Since these two sets of domains are legitimate and do not contain fast-flux domains, they can be used as a begin set to instantiate the weights. At first, we performed two consecutive DNS lookups of all domains. This lookup process took the TTL of each domain into account: we waited TTL + 1 seconds between two lookups to make sure not to get a cached

response from the nameserver in the second lookup. We repeated the lookup process several times.

In order to evaluate the detection performance of the proposed flux-score, we performed a 10-fold cross-validation on the corpus of labeled fast-flux and benign domains using different model parameters for finding the optimal hyperplane. The best model achieves an average detection accuracy of 99.98% with a standard deviation of 0.05%, thus almost no predictions on the testing data sets are incorrect. Regarding the weight vector w and bias b, the obtained assignments yield the following definition of the flux-score based on our experiments:

$$\begin{aligned} f(x) &= 1.32 \cdot n_A + 18.54 \cdot n_{ASN} + 0 \cdot n_{NS} \\ &\text{with } b = 142.38 \end{aligned} \tag{6.2}$$

Note, that the weight corresponding to n_{NS} is 0 and does not contribute to the detection of current FFSNs. Even though the flux-score is constructed from only two observed parameters, evading detection is difficult as the involved parameters n_A and n_{ASN} reflect essential properties of the underlying distributed structure of a FFSN.

The values of w_1, w_2 and w_3 as well as the threshold should be adjusted periodically since attackers could try to mimic CDNs in order to evade our metric, e.g., by sorting the IP addresses from their flux-agents according to IP address and then return only sequences of IP addresses that look like CDNs. We claim however that due to the two restrictions described above, it is hard for scammers to mimic *exactly* the behavior of a CDN. A fundamental difference between FFSNs and CDNs remains: a FFSN is built on top of compromised machines and the attacker has only limited influence on the availability, e.g., the user of the compromised machine can turn on or off the machine at arbitrary times. As part of future work, we want to examine how we can build a metric that automatically adapts to changes in FFSNs. This could for example implicitly include the fluxiness φ since φ for benign domains reaches its saturation limit pretty quickly comparing to fast-flux domains which have a growing fluxiness over time. In particular, benign domains with only one fixed IP have a constant φ $(= 1)$ from the very beginning of repeated DNS lookups. We would sacrifice our fast detection metric (only two DNS lookups are required now), but could possibly also detect more stealthy FFSNs. In a related work, Passerini et al. [PPMB08] introduced a different metric which employs several additional parameters. All of their parameters are also included in our metric, however, they include some parameters that contain slightly redundant information, e.g., *Number of distinct networks*, *Number of distinct autonomous systems* and *Number of distinct assigned network names* are three parameters that are closely related to each other. Their use of domain age and domain registrar could be included in our metric to improve the resistance against attacks, though. We will examine different methods to improve the metric as part of future work on this topic.

6.5 Tracking Fast-Flux Service Networks

Based on the information presented in the previous sections, we can now describe how to track fast-flux service networks. Please note that the mechanism behind this kind of networks differs significantly from malicious remote control networks as described in Chapters 4 and 5: the infected machines are not used to send out spam mails or to perform distributed denial-of-service attacks, but only to act as proxies and to relay HTTP requests to a central server. The infiltration step is thus a bit different compared to the tracking of botnets: on the one hand, we can *actively* infiltrate the network by executing a flux-agent in a controlled honeypot environment. We can then study the network from the inside and for example observe incoming requests and the actual proxy request to the mothership. This approach enables us to identify the mothership in an automated and efficient way. On the other hand, we can also *passively* infiltrate the network. In this case, we monitor the domain names used by the FFSNs and periodically perform a DNS lookup. This allows us to keep track of the flux-agents currently being used by the attackers to proxy content to the central server. We can monitor the dynamics of these networks and get an insight from a completely different angle. In the following, we show how such a passive infiltration can be implemented in practice and provide empirical measurement results.

6.6 Empirical Measurements on Fast-Flux Service Networks

We now present results of studies on fast-flux domains and the underlying scam infrastructure. This is the first empirical study of the fast-flux phenomenon giving details about FFSNs we observed during a two-month period in July/August 2007. Most important, we demonstrate that some results of a previous study in this area [AFSV07] have changed, since scammers now operate differently because they adopted FFSNs and use this technique to host their scams. Even in the short period since the results of that study, there is already a change in tactic by the scammers.

6.6.1 Scam Hosting via Fast-Flux Service Networks

In this section, we focus on how FFSNs are used by spammers, e.g., for hosting websites that offer pharmaceutical products or replica watches. We study the domains found in spam e-mails (*spamvertized* domains). There are commonly both fast-flux and benign domains in this set: a spammer could host the online shop for his scam on a normal, benign web server or use a FFSN to have a more reliable hosting infrastructure which is hard to take down and mitigate.

FFSNs Used by Spammers. Our study is based on a spam corpus we obtained from `http://untroubled.org/spam/`. The corpus contains 22,264 spam mails collected in August 2007 with the help of spamtraps, thus we can be sure that our corpus contains only spam messages. From all these mails, we were able to extract 7,389 unique

domains that were advertised in the spam mails. We performed two *dig* lookups on all these domains and computed the flux-score according to Equation 6.2. In total, we could identify 2,197 (29.7%) fast-flux domains in the corpus. Anderson et al. used in their study a spam corpus collected in November 2006, and they did not find any FFSNs [AFSV07]. They found that 6% of scams were hosted on multiple IP addresses, with one scam using 45. All the scams hosted on multiple IP addresses could be FFSNs, which were not identified as such.

The two *dig* lookups for all fast-flux domains identified in our spam corpus resulted in 1,737 unique IP addresses pointing to compromised machines running flux-agents. This demonstrates that FFSNs are a real threat and nowadays commonly used by attackers. By performing a reverse DNS lookup, we confirmed that the flux-agents are commonly located in DSL/dial-up ranges, thus presumably belong to inexperienced users.

The majority of the fast-flux domains (90.9%) consist of three domain-parts. For these, the third-level domain is usually a wildcard that acts as a CNAME (*canonical name*) for the second-level domain. To exclude wildcards, we only take the top- and second-level domain into account: in total, we can then identify 563 unique fast-flux domains. Only four top-level domains are targeted by attackers: `.com` was used 291 times (51.7%), `.cn` 245 times (43.5%), `.net` 25 times (4.4%), and `.org` twice (0.4%). Together with Nazario we performed a similar analysis for the period between late January 2008 and end of May 2008 and found a wider distribution of affected top-level domains [NH08].

Similarity of Scam Pages. We also studied how many scams are hosted via these FFSNs. Similar to a previous study [AFSV07], we want to explore the infrastructure used in these fraud schemes. We thus downloaded a snapshot of the webpage of each IP addresses returned in the *dig* lookup. The retrieved webpages comprise various dynamic content such as sessions numbers or randomized hostnames. These random strings, however, render analysis of content common to multiple webpages at the same IP address difficult. To tackle this issue we apply so called *string kernels*: a comparison method for strings, which is widely used in bioinformatics to assess similarity of DNA sequences [LEN02, STC04]. Compared to *image shingling* [AFSV07], a graphical method to find similar webpages, string kernels are more resilient to obfuscations: for example, we found several webpages that simply changed the background image and such similar sites cannot be identified via image shingling. Furthermore, string kernels enable comparison of documents with linear run-time complexity in the number of input bytes and yield performance rates up to 5,000 comparisons per second [RLM06, SRR07]. Moreover, as our approach considers the original HTML documents, no further preprocessing or rendering of HTML content is required. Thus using a string kernel approach to detect similarity of scam pages is significantly faster than image shingling.

For our setup, we employ a string kernel that determines the similarity of two webpages by considering n-grams (substrings of n bytes) shared by both pages. Given webpages p_1, p_2 and an n-gram a shared by p_1 and p_2, we first define $\phi_a(p_1)$ and

$\phi_a(p_2)$ as the number of occurrences of a in p_1 and p_2, respectively. The string kernel is then defined over the set A of all shared n-grams as

$$k(p_1, p_2) = \sum_{a \in A} \phi_a(p_1) \cdot \phi_a(p_2)$$

Note, that $k(p_1, p_2)$ corresponds to an inner-product in a vector space, whose dimensions are enumerated by all possible n-grams. Since $\phi_a(p_1)$ and $\phi_b(p_2)$ are natural numbers k is not bounded, so that we need to normalize it by considering a normalized variant $\hat{k}$

$$\hat{k}(p_1, p_2) = \frac{k(p_1, p_2)}{\sqrt{k(p_1, p_1) \cdot k(p_2, p_2)}}$$

The output of the kernel $\hat{k}$ is in the range 0 to 1, such that $\hat{k}(p_1, p_2) = 0$ implies that no shared n-grams exists and $\hat{k}(p_1, p_2) = 1$ indicates equality of p_1 and p_2. For all other cases $0 < \hat{k}(p_1, p_2) < 1$, the kernel $\hat{k}$ can be used as a metric to measure of similarity between webpages.

Grouping of Webpages. Using string kernels, we can define an intuitive method for determining groups of similar webpages located at a given IP address. For each address we compute a matrix K, whose entries K_{ij} correspond to kernel values $\hat{k}(p_i, p_i)$ of the i-th and j-th webpage. We then define a similarity threshold $0 < t < 1$ and assign two webpages p_i and p_j to the same group if $k(p_i, p_j) > t$ holds. Computing these assignments for all pages is carried out by simply looping over the columns of the matrix K. In empirical experiments we found a value of $k = 0.85$ to be a good threshold using the string kernel method for grouping webpages into scam hosts with the same content based on our data set.

Figure 6.6 shows the distribution of retrieved webpages per flux-agent. Please note that the x-axis is grouped into bins that grow quadratically. A little more than 50% of the flux-agents host just one webpage, but several pages per IP are not uncommon.

In Figure 6.7, we depict the distribution of unique scams hosted on one particular IP after having applied the grouping algorithms. We see that commonly attackers just proxy one scam via one flux-agent, but in 16.3% of our observations, also multiple scams are proxied through one particular flux-agent. This is significantly less than in the study performed by Anderson et al. [AFSV07], who found that 38% of scams were hosted on machines hosting at least one other scam. This indicates that scammers can now have a broader distribution of their infrastructure due to FFSNs.

6.6.2 Long-Term Measurements

To study the long-term characteristics of FFSNs, we observed several of them over a longer period of time: for 33 fast-flux domains found in spam e-mails, we performed a DNS lookup every 300 seconds over a period of seven weeks. In total, we observed 18,214 unique IP addresses during the measurement period between July 24 and September 10, 2007. This confirms that FFSNs are a real threat, with thousands of

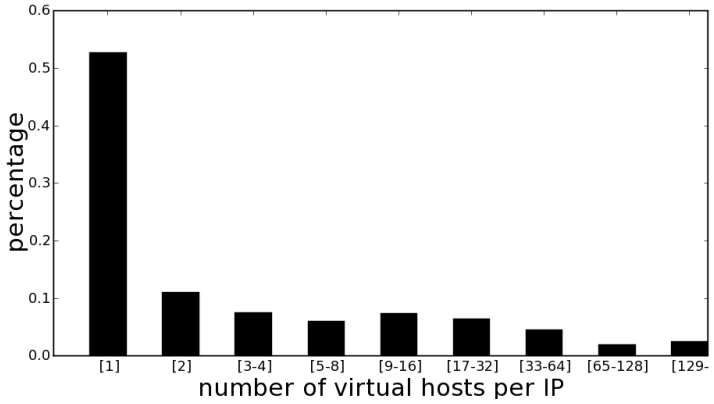


Figure 6.6: Distribution of virtual hosts per IP address per flux-agent

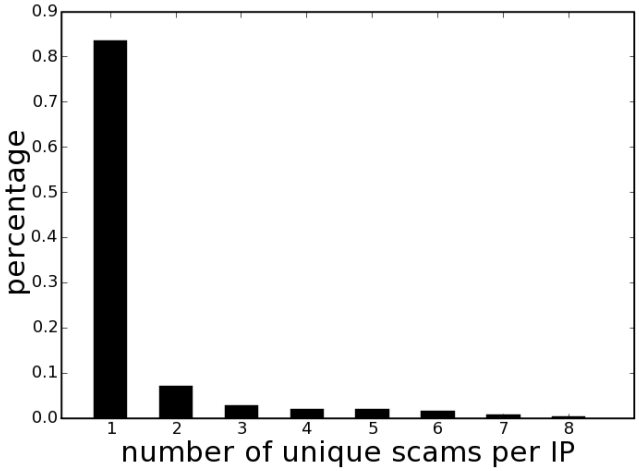


Figure 6.7: Distribution of unique scams per IP address per flux-agent

machines being compromised and abused. The monitored IPs belong to 818 unique ASNs and Table 6.2 lists the top eight ASNs we found. We see a wide distribution of flux-agents all over the (networked) world. Furthermore, the distribution follows a long-tail distribution, with 43.3% of all IPs contained in the top 10 ASNs.

This measurement does not take churn effects caused by DHCP or NAT into account. Note that NAT is no problem since a flux-agent needs to be reachable in order to serve as content proxy. We estimate that the percentage of churn caused by DHCP is rather small: in order to be a reliable flux-agent, the machine should be online for a longer time as otherwise it could cause downtime for the scam infrastructure. Thus an attacker will make sure to only include *stable* nodes, i.e., nodes that have a high uptime and constant IP address, into the pool of IP addresses served within FFSNs.

#	ASN	AS name and country	# observed flux-agents
1)	7132	AT&T Internet Services, US	2,677
2)	9304	Hutchison Global, HK	1,797
3)	4766	Korea Telecom, KR	590
4)	3320	Deutsche Telekom, DE	500
5)	8551	Bezeqint Internet, IL	445
6)	12322	Proxad/Free ISP, FR	418
7)	8402	Corbina telecom, RU	397
8)	1680	NetVision Ltd., US	361

Table 6.2: Top eight ASNs observed while monitoring 33 fast-flux domains over a period of seven weeks. The table includes the name and country of the AS, and the number of fast-flux IPs observed in this AS.

For each of the 33 FFSN domains, we examined the *diversity* of returned A records: each time we observe a new A record, we assign an ascending ID to this IP address. This allows us to keep track of how often and when we have seen a particular IP. The upper

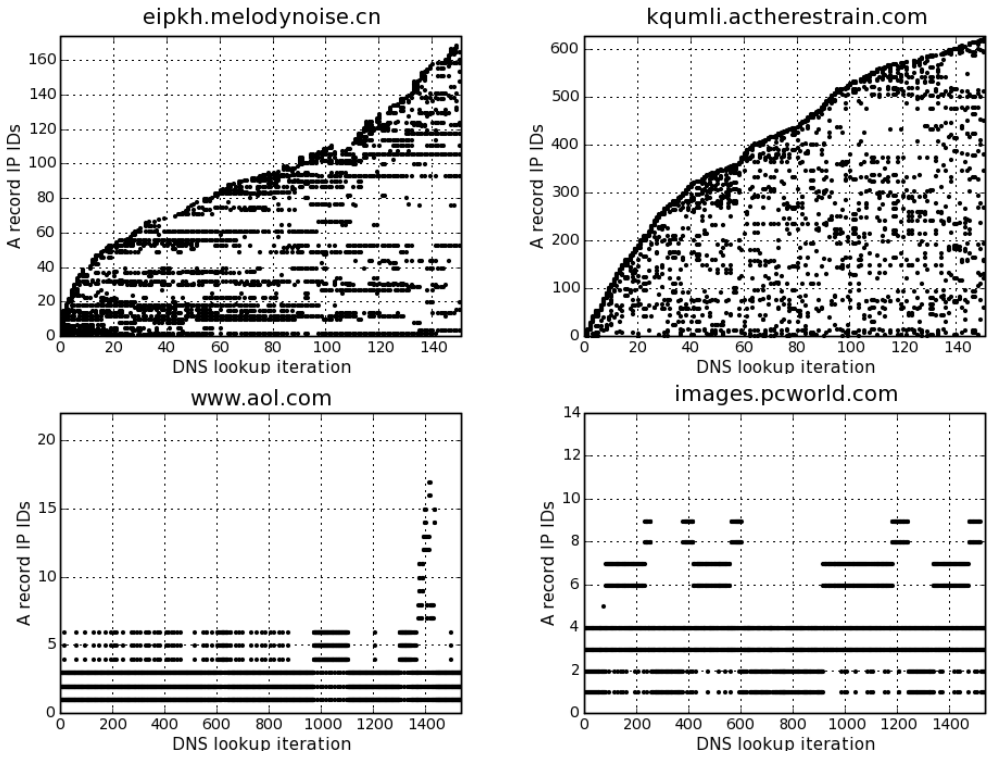


Figure 6.8: IP address diversity for two characteristic fast-flux domains (upper part) and two domains hosted via CDNs (lower part).

part of Figure 6.8 plots the diversity of IPs for a period of 12.5 hours for two exemplary fast-flux domains. We see a wide variety of IPs returned in our DNS lookups and a steady increase of new fast-flux IPs we monitor (leading to an increase in the fluxiness φ). The slope of both graphs is different, indicating that different FFSNs have a different value for φ. Furthermore, this graph also highlights the dimension of flux-agents: within the short amount of time (150 lookups), more than 600 unique flux-agents were returned for the fast-flux domain shown in the upper part of the figure.

In contrast we also plot IP diversity over time for two benign domains in the lower part of Figure 6.8. Please note that the measurement period for benign domains is ten times more DNS lookups to show some effects in the plot. For the CDN domains, we observe only a small total number of unique IP addresses returned and a clear pattern.

We found the fluxiness φ to be a reliable feature in case of many repeated DNS lookups: even though it grows for both CDNs and fast-flux domains during the first few rounds of DNS lookups, a saturation can be seen earlier for the CDNs and hence we can reliably decide whether or not a given domain uses fast-flux techniques after repeated lookups by only considering the number of unique IP addresses observed during lookups, i.e., n_A.

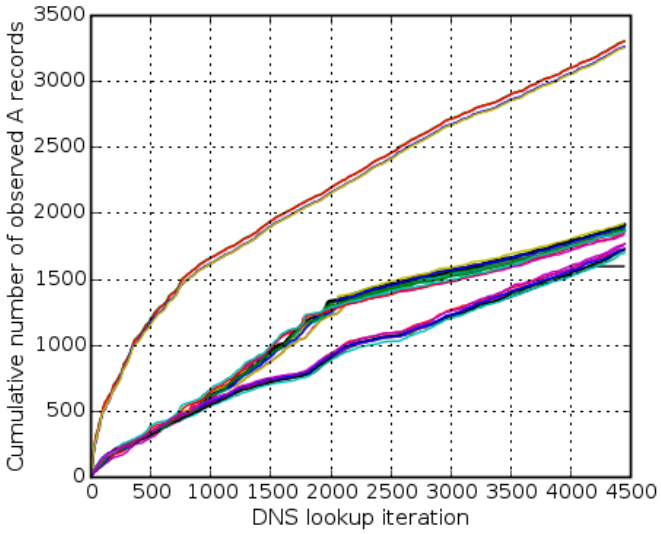


Figure 6.9: Cumulative number of distinct A records observed for 33 fast-flux domains.

To further study the long-term growth of n_A and n_{ASN}, we present in Figure 6.9 the cumulative number of distinct A records and in Figure 6.10 the cumulative number of ASNs observed for each of the 33 fast-flux domains during a period of more than 15 days. We see three different classes of growth in both figures. This means that different fast-flux domains have a characteristic distribution of flux-nodes. If two domains have a similar growth of the cumulative number of distinct A records or ASNs, this could indicate that both domains belong to the same FFSN: the nameservers return A records

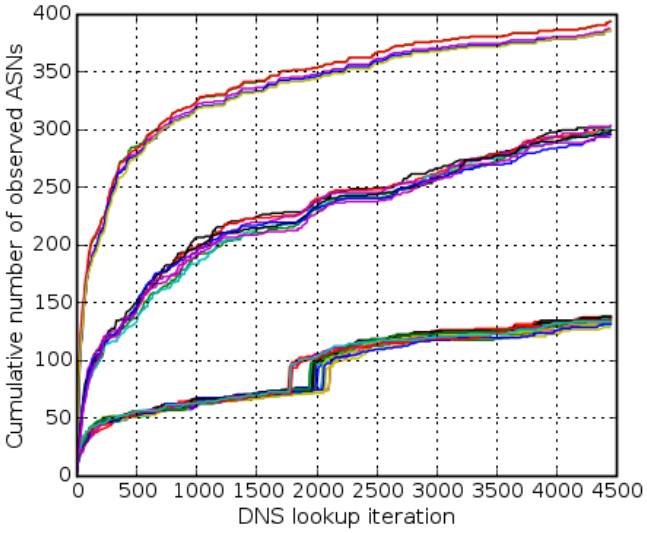


Figure 6.10: Cumulative number of distinct ASNs observed for 33 fast-flux domains.

with similar characteristics. We plan to examine this in the future as a possible way to identify different domains belonging to the same control infrastructure. Furthermore, the two figures also show a declining growth over time. A longer measurement of the number of distinct A records or ASNs could be used to estimate the overall size of the pool of compromised machines used for a particular FFSN: since the curves will eventually reach a saturation, this is an upper bound of the pool size. The initial strong growth in Figure 6.10 also indicates that flux-agents are commonly spread over several ASs, confirming our weighting for the flux-score.

One goal of FFSNs is to provide a robust hosting infrastructure for cybercriminals. However, we found that FFSNs also need to deal with unavailability of the site, especially caused by the unavailability of the DNS server itself. From the 374,427 DNS queries during the measurement period, 16,474 (4.60%) failed. We also monitored 16 legitimate domains from the Alexa Top 500 to measure the reliability of benign domains. From 128,847 lookups against these domains, only 17 (0.01%) failed.

6.6.3 Other Abuses of Fast-Flux Service Networks

Besides using FFSNs to host scam sites related to spam, we also found several other illegal use cases for these networks. This is presumably due to the fact that FFSNs provide a robust infrastructure to host arbitrary content: they are not restricted to work with HTTP servers, but an attacker could also set up a fast-flux SMTP or fast-flux IRC server. In this section, we briefly provide information about two additional examples of how attackers use FFSNs as part of their infrastructure.

First, fast-flux networks are commonly used by phishing groups. *Rock phish* is a well-known phishing toolkit which allows an attacker to set up several phishing scams in parallel: the attacker installs the phishing kit on a webserver and different URL-paths lead to different phishing pages [MC07]. The actual domain belonging to these phishing pages commonly uses fast-flux service networks to host the website. For example, the domain `regs26.com` was used for phishing in September 2007. By performing a DNS lookup every time the TTL expired, we observed a total of 1,121 unique A records during a measurement period of four days.

Second, also botherders use FFSNs to host malicious content. The Storm Worm botnet, which we covered in detail in Chapter 5, uses fast-flux domains to host the actual bot binary. We monitored the domain `tibeam.com` for a period of four weeks in August 2007 and observed more than 50,000 unique IP addresses in the returned A records for this particular domain. This indicates that Storm Worm is a large botnet, since the pool of IP addresses served for the FFSNs is apparently large. Furthermore, monitoring the domain allows us to keep track of the botnet: since each compromised machine also runs a webserver, which hosts the actual bot binary, we can download the current binary from this host.

6.7 Mitigation of Fast-Flux Service Networks

In this section, we briefly review several strategies to mitigate the threat posed by FFSNs. Even though the client's view onto an FFSN is pretty limited (we can only monitor the flux-agents), we try to collect as much information as possible with the techniques outlined in the previous sections. Our metric helps us to automatically find fast-flux domains, which can be collected in a *domain blacklist*. First, such a blacklist can be used to stop a fast-flux domain with the help of collaboration from domain name registrars: a registrar has the authority to shut down a domain, thus effectively taking down the scam. An automated blacklist of fast-flux domains can quickly notify registrars about fraudulent domains. Second, an ISP can use such a blacklist to protect its clients from FFSNs by blackholing DNS requests for fast-flux domains. Third, the domain blacklist can be used for spam filtering: if an e-mail contains a fast-flux domain, it is most likely a spam mail. This technique could be used at the client- or server-side if slight delay is tolerable. Tracking of FFSNs by periodically performing DNS lookups for fast-flux domains can be used to build a list of IPs which most likely are compromised, which could be used in a similar way as the domain blacklist.

Similar to an anonymity system, a FFSN has one layer of redirection: a client cannot directly observe the location of the control node, which hosts the actual content, but only the flux-agent. Ideas from this area can be adopted to identify the actual location of the control node: an ISP has the capability to monitor "from above" both the incoming and outgoing flows for a given machine and thus can monitor flows belonging to a flux-agent. If a flux-agent is located within the network of an ISP, the idea is to inject requests which are proxied from the agent to the control node (step 2 in Figure 6.5).

Together with the response to such requests, the location of the control node can be identified and this content-serving central host can then be taken down [The07].

As FFSNs might be only the first step towards high available scams, we should also think of more general approaches on combating this kind of distributed, malicious infrastructure. One possibility would be to block certain incoming connection requests directed to dial-up ranges, e.g., to TCP port 80 or UDP port 53. The majority of dial-up users does not need to host servers, and such an approach would block many possibilities to abuse compromised clients. ISPs could change their policy to not allow any network services within mainly dynamic IP ranges by default. Still certain ports could be enabled by whitelisting if there is a need for network services for specific users.

6.8 Summary

In this chapter, we showed how a different kind of malicious remote control networks, namely fast-flux service networks, can also be studied with our methodology. We presented the first empirical study of FFSNs, which use compromised host to build a proxy network that is then used to host content. In this kind of attack, the control network is *indirect*, i.e., the attacker does not send directly commands to the infected machines, but abuses their network resources of the victim to establish a proxy network. This is in contrast to classical botnets which we studied in the previous chapter, since these networks use a *direct* command channel. Nevertheless, we can still track this kind of networks with our method and learn more about them in an automated way.

For the study, we developed a metric that exploits the principles of FFSNs to derive an effective mechanisms for detecting new fast-flux domains in an automated way. Beside being straightforward to compute, we also showed that the method is accurate, i.e., we had very low false positive and false negative rates. As we are aware of the dynamics within fast-flux, we expect the need of further refinements in our method. Based on our empirical observations, we found other information, e.g., `whois` lookups and MX records, as promising features for an extended version of our flux-score.

Beside analyzing FFSN features in terms of detection and mitigation, we plan to work on statistical methods to estimate how many IP addresses are in a certain pool of one fast-flux domain. Adopting *capture-recapture* methods, which are applied in biology for measuring the number of members of a certain population [PNBH90], could be one way to obtain such an estimation. Similar methods were successfully applied by Weaver and Collins to measure the extent of phishing activity on the Internet [WC07] and we plan to study how this technique could be adopted for the area of fast-flux service networks to estimate the number of infected machines.

Chapter 7

Conclusion and Future Work

With the current operating systems and network designs, an attacker can often compromise a victim's machine and obtain complete control over this machine without too much effort. On the one hand, an attacker can for example use technical means like a remote exploit to take advantage of a vulnerable network service, or utilize a vulnerability on a client application to compromise a system. On the other hand, she can also use social means like social engineering, e.g., tricking the victim into opening a malicious e-mail attachment, to attack a user. Based on these kinds of attacks, she can compromise a large number of machines and establish a malicious remote control network. We expect that this kind of networks will be used by attackers in the next few years since no effective countermeasures against all kinds of threats can be expected in the near future. Especially social engineering is still a growing problem, with many users falling for this kind of scam. And attackers increasingly take advantage of the compromised machines, e.g., by using them to carry out distributed denial-of-service attacks, to send spam mails, or other means to take financial advantage.

To combat these attacks, we need to develop effective countermeasures against malicious remote control networks and the factors behind them. These countermeasures can apply on different levels. For example, many different methods have been proposed to stop or circumvent memory corruption attacks (see for example the work by Cowan et al. [CPM+98], Shankar et al. [STFW01], Cowan et al. [CBJW03], or Ruwase et al. [RL04]), thereby stopping the most prevalent attack vector. Another approach is to address the financial motivation behind these attacks [FG06, LLS08] and making these attacks unprofitable. Increasing user awareness by for example teaching them about current attacks [KGOL05] is a third approach to combat these attacks.

In this thesis, we proposed a methodology to deal with malicious remote control networks by providing a root cause approach to stop this kind of networks. The basic idea of the method is to first infiltrate the network, then analyze the network from the inside, and finally use the collected information to stop it. This addresses the problem of these networks in a reactive way, offering the possibility to stop them once they have been established. Such an approach is necessary since we anticipate that attackers will

always find novel ways to compromise a larger number of machines and use them for nefarious purposes. In a sense, this approach seems to be always one step behind the attacker. However, by automating the method to a high degree we are able to compete against the attackers and learn more about their methods. Also note that our method neither implies a resource arms race nor needs any additional infrastructure.

In the following, we briefly summarize the important results of this thesis and outline directions for future research in the individual areas.

Chapter 3: Root-Cause Methodology to Prevent Malicious Remote Control Networks. In Chapter 3 we introduced the general methodology to stop malicious remote control networks. The method is based on the insight that an attacker usually wants to control a large number of compromised machines such that she can perform a DDoS attack or send out a many spam mails. To coordinate a large number of machines, she needs a remote control mechanism. Therefore we can stop the whole attack if we are able to disable the remote control mechanism in an efficient way.

In the following chapters we showed that this method can be applied to different kinds of malicious remote control networks that exist today. In the future, we want to extend this study and examine what other kind of attacks can be stopped by the same approach. An obvious extension is to use this method to study *spambots*, i.e., bots that focus on sending spam mails only, in detail [JMGK09].

Chapter 4: Tracking Botnets with Central C&C Server. In the first case study, we focussed in Chapter 4 on botnets with a central command and control (C&C) server. In this setup, the attacker uses a central server to disseminate the commands to the infected machines. This is the classical type of botnets prevalent today and thus an important problem to address.

As outlined above, stopping malicious remote control networks requires several steps. Therefore we first showed how to use honeypots to obtain samples of autonomous spreading malware in an automated way. In the second step, we introduced a method to analyze a given sample in an automated way by performing a dynamic, runtime analysis. Finally, we used the obtained information to impersonate as a bot and join the network. We can then start observing the network from the inside and collect detailed information about the setup, which can then be used to stop the botnet. This method is used by several organizations. For example, the ShadowServer Foundation, a group of security professionals that gathers, tracks, and reports on malware, botnet activity, and electronic fraud [Sha09], and many other groups around the world use the tools and methods outlined in Chapter 4 to combat botnets.

We expect that botnets with a central C&C server will prevail for several years since this kind of control over compromised machines is the easiest from an attacker's point of view. Thus we need to continue studying this kind of botnets and examine whether or not the method needs to be adjusted in the future. Custom protocols for C&C communication, encrypted or hidden communication channels, or similar stealth bots will certainly provide opportunities for future research in this area.

Chapter 5: Tracking Botnets with Peer-to-Peer-based C&C Server. In Chapter 5 we analyzed how botnets with a peer-to-peer-based C&C channel can be tracked with our methodology. We showed how to successfully apply the proposed method to Storm Worm, the most prevalent bot using a decentralized communication protocol observed until now. The method is similar to the case of botnets with a central server, we just had to adjust the observation step by implementing a crawler that can be used to contact all bots within the network.

In the future, we expect the emergence of more advanced malicious remote control networks that could defeat current defense mechanisms such as the ones proposed in this thesis. To increase the understanding of such networks, we introduced the design of an advanced botnet named *Rambot* [HHH08]. The design is based on the weaknesses we identified when studying different kinds of malicious remote control networks during the work on this thesis. The main features of this bot are peer-to-peer communication, strong cryptography, a credit-point system to build bilateral trust amongst bots, and a proof-of-work scheme to protect against potential attacks. While some of our design decisions seem to be obvious, e.g., using strong cryptography, (almost) no current botnet uses them. Furthermore, some aspects of Rambot's design are countermeasures against new techniques to track botnets, e.g., proof-of-work schemes to counter crawling of the whole botnet (see Chapter 5 for details). The goal of this discussion is a better understanding of how next-generation malicious remote control networks could be developed by attackers and what to do against them.

Starnberger et al. recently introduced *Overbot*, a botnet communication protocol based a peer-to-peer architecture [SKK08]. Their design also demonstrates the threats that may result when future botnets utilize more advanced communication structures. We anticipate that such discussions help understanding future attacks, such that we can begin to develop better countermeasures against novel threats. Therefore it is crucial to also think about future malicious remote control networks and how their design could look like. For example, the communication pattern used by NNTP [KL86] is much more suitable for anonymous communication than distributed hash table (DHT) based routing. This communication pattern is the core of fault-tolerant information dissemination like in Golding's *weak consistency replication* [Gol92] or the reliable broadcast algorithms by Hadzilacos and Toueg [HT93]. Botnets based on these design principles and techniques will be much harder to mitigate than the current ones.

Chapter 6: Tracking Fast-Flux Service Networks. We studied in Chapter 6 a completely different kind of malicious remote control networks, namely fast-flux service networks. The idea behind these networks is that the attacker does not directly abuse the compromised machines, but uses them to establish a proxy network on top of these machines to enable a robust hosting infrastructure.

We proposed a metric to identify fast-flux service networks (FFSNs) in an automated way. The metric is based on the general principles of these networks, e.g., an attacker is not free to choose the hardware and network location of an individual node (*IP address diversity*) and the nodes can go down at arbitrary times (*no physical agent control*).

These restrictions were used to find distinguishing parameters which enable us to decide in an efficient way whether or not a given domain uses the technique of fast-flux or not. We need to carefully study how attackers could subvert our metric and examine how a more robust metric could be developed. Furthermore, our understanding of fast-flux service networks is rather limited up to now, thus we need to closely study this phenomenon to learn more about the underlying techniques.

Other Directions of Future Work. We presented several honeypot tools that can be used to study current attacks on the Internet. Since the behavior of attackers is constantly changing and new attack vectors constantly evolve, also the honeypot tools need to advance. A first step in this area is the observation that many novel attacks target client applications instead of network services. As a result, several different types of client-side honeypots have been developed [WBJ+06, SS09, Wan09]. How to study attacks in IPv6 networks, or against the network backbone and infrastructure is still an open problem. Furthermore, targeted attacks and spear phishing cannot easily be observed with current honeypot-based techniques. Thus there are many options for future work in the area of honeypots which could address the development of new tools and techniques to track latest attacks.

In addition, we presented an approach for automated, dynamic analysis of malware in this thesis. CWSandbox is nowadays a commercial tool and used by many organizations worldwide. More than one million samples have been analyzed with a CWSandbox installation at the Laboratory for Dependable Distributed Systems and many gigabytes of analysis results have been generated by this system. Using this data for novel applications is an interesting area for future research. For example, such data can be used to perform malware classification [RHW+08] or malware clustering [BOA+07, BMH+09]. Improving these results and using the outcome of the algorithms, e.g., by generating behavioral signatures for the individual clusters, are topics of future work in this area.

The Internet Malware Analysis System (*Internet-Malware-Analyse-System, InMAS*) is a distributed system to collect and analyze malware. The project is developed at the Laboratory for Dependable Distributed Systems and funded by the German Federal Office for Information Security (*Bundesamt für die Sicherheit in der Informationstechnik, BSI*). The core functionality of InMAS is based on honeypots as well as automated analysis tools such as *CWSandbox* (see Chapter 4). InMAS is being developed as a building block for an early-warning system such that autonomously spreading malware in its many forms can be detected and analyzed in detail. InMAS also allows to connect different data-sources so that analysis and detection can be further improved. The result of this research project is a unique set of collection and analysis tools for autonomously spreading malicious software. Statistics and analysis results enable the operator of the early-warning system to detect, classify, and react to the network threats caused by such malware. The data can for example be used to trace back the origin of the outbreak of a worm, or to answer questions regarding the network propagation of autonomous spearing malware over time.

Bibliography

[AFSV07] David S. Anderson, Chris Fleizach, Stefan Savage, and Geoffrey M. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[Aka] Akamai Technologies. Akamai Content Distribution Network. `http://www.akamai.com`.

[Ale07] Alexa, the Web Information Company. Global Top 500 Sites, September 2007. `http://alexa.com/site/ds/top_sites?ts_mode=global`.

[ASA+05] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos P. Markatos, and Angelos D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[Bal07] Josh Ballard. Storm Worm, October 2007. NANOG 41, `http://www.nanog.org/mtg-0710/kristoff.html`.

[Bar96] Dave Barr. RFC 1912: Common DNS Operational and Configuration Errors, February 1996. `http://www.ietf.org/rfc/rfc1912.txt`.

[BCH06] J. Black, M. Cochran, and T. Highland. A Study of the MD5 Attacks: Insights and Improvements. In *Fast Software Encryption*, 2006.

[BCJ+05] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, 2005.

[Bel01] Steve M. Bellovin. ICMP traceback messages, March 2001. Internet Draft.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[BH06] Rainer Böhme and Thorsten Holz. The Effect of Stock Spam on Financial Markets. In *Proceedings of 5th Workshop on the Economics of Information Security (WEIS'06)*, June 2006.

[Bin06] James R. Binkley. Anomaly-based Botnet Server Detection. In *Proceedings of FloCon 2006 Analysis Workshop*, October 2006.

[Bir04] Alessandro Birolini. *Reliability Engineering: Theory and Practice*. Springer Verlag, 2004.

[BKH+06] Paul Bächer, Markus Kötter, Thorsten Holz, Felix Freiling, and Maximillian Dornseif. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proceedings of 9th International Symposium On Recent Advances in Intrusion Detection (RAID'06)*, September 2006.

[BKK06] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR'06)*, 2006.

[BM00] P.S. Bradley and O.L. Mangasarian. Massive Data Discrimination via Linear Support Vector Machines. *Optimization Methods and Software*, 13:1–10, 2000.

[BMH+09] Ulrich Bayer, Paolo Milani, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, February 2009.

[BMKK06] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2:67–77, 2006.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of 10th International Symposium On Recent Advances in Intrusion Detection (RAID'07)*, 2007.

[Bri95] Thomas P. Brisco. RFC 1794: DNS Support for Load Balancing, April 1995. `http://www.ietf.org/rfc/rfc1794.txt`.

[BS06] James R. Binkley and Suresh Singh. An Algorithm for Anomaly-based Botnet Detection. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI'06)*, pages 43–48, July 2006.

[BV05] Edward Balas and Camilo Viecco. Towards a Third Generation Data Capture Architecture for Honeynets. In *Proceeedings of the 6th IEEE Information Assurance Workshop*, West Point, 2005.

[BY07] Paul Barford and Vinod Yegneswaran. *An Inside Look at Botnets*, volume 27 of *Advances in Information Security*, pages 171–191. Springer, 2007.

[Can05] John Canavan. The Evolution of Malicious IRC Bots. In *Proceedings of the Virus Bulletin Conference*, 2005.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[CCY99] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[Che06] Yan Chen. IRC-Based Botnet Detection on High-Speed Routers. In *ARO-DARPA-DHS Special Workshop on Botnets*, June 2006.

[CJM05] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI'05)*, pages 39–44. USENIX, June 2005.

[Com96] Computer Emergency Response Team. CERT advisory CA-1996-21 TCP SYN Flooding Attacks. Internet: `http://www.cert.org/advisories/CA-1996-21.html`, 1996.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[Cui06] Weidong Cui. *Automating Malware Detection by Inferring Intent*. PhD thesis, University of California, Berkeley, September 2006.

[Cym06] Team Cymru: The Darknet Project. Internet: `http://www.cymru.com/Darknet/`, Accessed: 2006.

[DHK04] Maximillian Dornseif, Thorsten Holz, and Christian Klein. NoSEBrEaK – Attacking Honeynets. In *Proceedings of the 5th IEEE Information Assurance Workshop*, West Point, 2004.

[Dit09] Dave Dittrich. Distributed Denial of Service (DDoS) attacks/tools resource page. Internet: `http://staff.washington.edu/dittrich/misc/ddos/`, Accessed: March 2009.

[Dou02] John R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, March 2002.

[DZL06] David Dagon, Cliff Zou, and Wenke Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)*, 2006.

[Enr07] Brandon Enright. Exposing Stormworm, October 2007. Toorcon 9, `http://noh.ucsd.edu/~bmenrigh/`.

[Fat04] Holy Father. Hooking Windows API – Technics of Hooking API Functions on Windows. *Code Breakers Journal*, 1(2), 2004.

[Fed04] Federal Bureau of Investigation (FBI). Report on Operation Cyberslam. Internet: `http://www.reverse.net/operationcyberslam.pdf`, February 2004.

[Fed06] Federal Bureau of Investigation (FBI). Moroccan Authorities Sentence Two In Zotob Computer Worm Attack. `http://www.fbi.gov/pressrel/pressrel06/zotob091306.htm`, September 2006.

[Fed07] Federal Bureau of Investigation (FBI). Operation Bot Roast, February 2007. `http://www.fbi.gov/pressrel/pressrel07/botnet061307.htm`.

[Fer00] Paul Ferguson. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, May 2000. Request for Comments: RFC 2827.

[FG06] Richard Ford and Sarah Gordon. Cent, Five Cent, Ten Cent, Dollar: Hitting Botnets Where it Really Hurts. In *Proceedings of the 2006 Workshop on New Security Paradigms (NSPW'06)*, pages 3–10, 2006.

[FHW05] Felix Freiling, Thorsten Holz, and Georg Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *Proceedings of 10th European Symposium On Research In Computer Security (ESORICS'05)*, July 2005.

[Fis05] Tom Fischer. Botnetze. In *Proceedings of 12th DFN-CERT Workshop*, March 2005.

[FPPS07] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of 14th Conference on Computer and Communications Security (CCS'07)*, November 2007.

[Fra07] Frank Boldewin. Peacomm.C – Cracking the Nutshell, September 2007. `http://www.reconstructer.org/`.

[Gar00] Lee Garber. Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, April 2000.

[GCR01] Syam Gadde, Jeffrey S. Chase, and Michael Rabinovich. Web Caching and Content Distribution: A View from the Interior. *Computer Communications*, 24(2):222–231, 2001.

[GH07] Jan Göbel and Thorsten Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In *Proceedings of Hot Topics in Understanding Botnets (HotBots'07)*, April 2007.

[GHH06] Jan Göbel, Jens Hektor, and Thorsten Holz. Advanced Honeypot-Based Intrusion Detection. *USENIX ;login:*, 31(6), December 2006.

[GHW07] Jan Göbel, Thorsten Holz, and Carsten Willems. Measurement and Analysis of Autonomous Spreading Malware in a University Environment. In *Proceedings of 4th Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'07)*, July 2007.

[Göb08] Jan Göbel. Amun: Python Honeypot. Internet: `http://amunhoney.sourceforge.net/`, October 2008.

[Gol92] Richard A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California at Santa Cruz, December 1992. UCSC- CRL-92-52.

[GPY+07] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, August 2007.

[GPZL08] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, 2008.

[Gro03] LURHQ Threat Intelligence Group. Sinit P2P Trojan Analysis. Internet: `http://www.lurhq.com/sinit.html`, 2003.

[Gro04a] LURHQ Threat Intelligence Group. Bobbax Worm Analysis. Internet: `http://www.lurhq.com/bobax.html`, 2004.

[Gro04b] LURHQ Threat Intelligence Group. Phatbot Trojan Analysis. Internet: `http://www.lurhq.com/phatbot.html`, 2004.

[Gro07] Lev Grossman. The Worm That Roared. Internet: `http://www.time.com/time/magazine/`, September 2007.

[GSN+07] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-Peer Botnets: Overview and Case Study. In *Proceedings of Hot Topics in Understanding Botnets (HotBots'07)*, 2007.

[Gu08] Guofei Gu. *Correlation-based Botnet Detection in Enterprise Networks*. PhD thesis, Georgia Institute of Technology, August 2008.

[GZL08] Guofei Gu, Junjie Zhang, and Wenke Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[HB99] Galen C. Hunt and Doug Brubacker. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143. Advanced Computing Systems Association, 1999.

[HEF08] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzone. Technical Report TR-2008-006, University of Mannheim, December 2008.

[Hew07] Hewlett Packard Inc. NonStop home page: HP Integrity NonStop computing. Online: `http://h20223.www2.hp.com/nonstopcomputing/cache/76385-0-0-225-121.aspx`, September 2007.

[HGRF08] Thorsten Holz, Christian Gorecki, Konrad Rieck, and Felix Freiling. Measuring and Detecting Fast-Flux Service Networks. In *Proceedings of 15th Annual Network & Distributed System Security Symposium (NDSS'08)*, February 2008.

[HHH08] Ralf Hund, Matthias Hamann, and Thorsten Holz. Towards Next-Generation Botnets. In *European Conference on Computer Network Defense (EC2ND'08)*, pages 33–40, December 2008.

[Hol05] Thorsten Holz. A Short Visit to the Bot Zoo. *IEEE Security & Privacy*, 3(3):76–79, 2005.

[Hol06] Thorsten Holz. Learning More About Attack Patterns With Honeypots. In *Proceedings of Sicherheit 2006*, pages 30–41, February 2006.

[Hon05] Honeynet Project. Know your Enemy: Tracking Botnets, March 2005. `http://www.honeynet.org/papers/bots`.

[HR05] Thorsten Holz and Frederic Raynal. Detecting Honeypots and Other Suspicious Environments. In *Proceeedings of the 6th IEEE Information Assurance Workshop*, West Point, June 2005.

[HSD+08] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *Proceedings of First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'08)*, April 2008.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

[IHF08] Ali Ikinci, Thorsten Holz, and Felix Freiling. Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In *Proceedings of Sicherheit 2008*, April 2008.

[Int07] Internet Software Consortium. dig: domain information groper, September 2007. `http://www.isc.org/sw/bind/`.

[Iva02] Ivo Ivanov. API Hooking Revealed. The Code Project, 2002.

[JMGK09] John P. John, Alexander Moshchuk, Steven D. Gribble, and Arvind Krishnamurthy. Studying Spamming Botnets Using Botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[JX04] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of 13th USENIX Security Symposium*, 2004.

[Kad] KadC. P2P library. `http://kadc.sourceforge.net/`.

[KAG06] Andrew Kalafut, Abhinav Acharya, and Minaxi Gupta. A Study of Malware in Peer-to-Peer Networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 327–332, 2006.

[KBM94] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA Prototype. In *Selected Papers of the First Conference on World-Wide Web*, pages 155–164. Elsevier Science Publishers B. V., 1994.

[KGOL05] Sven Krasser, Julian B. Grizzard, Henry L. Owen, and John G. Levine. The Use of Honeynets to Increase Computer Network Security and User Awareness. *Journal of Security Education*, 1(2):23–37, 2005.

[KKL+08a] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoff Voelker, Vern Paxson, and Stefan Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion . In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, October 2008.

[KKL+08b] Christian Kreibich, Chris Kanich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. On the Spam Campaign Trail. In *Proceedings of First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'08)*, 2008.

[KL86] Brian Kantor and Phil Lapsley. Network News Transfer Protocol (NNTP). Internet RFC 977, available at `http://www.faqs.org/rfcs/rfc977.html`, February 1986.

[Kre07] Brian Krebs. Storm Worm Dwarfs World's Top Supercomputers. Internet: `http://blog.washingtonpost.com/securityfix/`, August 2007.

[KRH07] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. Wide-Scale Botnet Detection and Characterization. In *Proceedings of the Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, April 2007.

[KWZ01] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the Use and Performance of Content Distribution Networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 169–182, 2001.

[LD08] Corrado Leita and Marc Dacier. SGNET: A Worldwide Deployable Framework to Support the Analysis of Malware Threat Models. In *Proceedings of 7th European Dependable Computing Conference (EDCC'08)*, 2008.

[LDM06] Corrado Leita, Marc Dacier, and Frédéric Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen Based Honeypots. In *Proceedings of 9th Symposium on Recent Advances in Intrusion Detection (RAID'06)*, 2006.

[Lei08] Corrado Leita. *SGNET: Automated Protocol Learning for the Observation of Malicious Threats*. PhD thesis, Eurecom, December 2008.

[LEN02] Christina Leslie, Eleazar Eskin, and William S. Noble. The Spectrum Kernel: A String Kernel for SVM Protein Classification. In *Proceedings of Pacific Symposium on Biocomputing*, pages 564–575, 2002.

[LLS08] Zhen Li, Qi Liao, and Aaron Striegel. Botnet Economics: Uncertainty Matters. In *Proceedings of the 7th Workshop on the Economics of Information Security (WEIS'08)*, 2008.

[LMD05] Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[Mai03] SecurityFocus Honeypots Mailinglist. Moving forward with defintion of honeypots, May 2003. Internet: `http://www.securityfocus.com/archive/119/321957/30/0/threaded`.

[Max] Maxmind. Geolocation and Online Fraud Prevention. `http://www.maxmind.com/`.

[MC07] Tyler Moore and Richard Clayton. An Empirical Analysis of the Current State of Phishing Attack and Defence. In *Proceedings of the 6th Workshop on the Economics of Information Security (WEIS'07)*, 2007.

[McC03a] Bill McCarty. Automated Identity Theft. *IEEE Security & Privacy*, 1(5):89–92, 2003.

[McC03b] Bill McCarty. Botnets: Big and Bigger. *IEEE Security & Privacy*, 1(4):87–90, 2003.

[MDDR04] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall PTR, 2004.

[Mea98] Catherine Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. In *Proceedings of the 1999 IEEE Computer Security Foundations Workshop*, pages 4–13. IEEE Computer Society Press, 1998.

[Mic03] Microsoft. Security Bulletin MS03-026: Buffer Overrun In RPC Interface Could Allow Code Execution, July 2003.

[Mic04] Microsoft. Security Bulletin MS04-011: Security Update for Microsoft Windows, April 2004.

[Mic05] Microsoft. Security Bulletin MS05-039: Vulnerability in Plug and Play Could Allow Remote Code Execution and Elevation of Privilege, August 2005.

[Mic08] Microsoft. Security Bulletin MS08-067: Vulnerability in Server Service Could Allow Remote Code Execution, October 2008.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*, 2007.

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer Informatiion System Based on the XOR Metric. In *Proceedings of the 1st Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[MPS+03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[MR04] Jelena Mirkovic and Peter Reiher. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[MRRK03] Jelena Mirkovic, Max Robinson, Peter Reiher, and Geoff Kuenning. Alliance Formation for DDoS Defense. In *Proceedings of the New Security Paradigms Workshop 2003*. ACM SIGSAC, August 2003.

[MSkc02] David Moore, Colleen Shannon, and k claffy. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 273–284, 2002.

[MSVS04] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Network Telescopes. Technical Report TR-2004-04, CAIDA, 2004.

[MT06] Jerry Martin and Rob Thomas. The Underground Economy: Priceless. *USENIX ;login:*, 31(6), December 2006.

[Müt07] Michael Müter. Web-based Honeypot Decoys. Master's thesis, RWTH Aachen University, Germany, April 2007.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[Nau07] John Naughton. In Millions of Windows, the Perfect Storm is Gathering. `http://observer.guardian.co.uk/`, October 2007.

[Naz06] Jose Nazario. Nugache: TCP port 8 Bot. Internet: `http://asert.arbornetworks.com/2006/05/nugache-tcp-port-8-bot/`, May 2006.

[Naz07] Jose Nazario. Estonian DDoS Attacks – A summary to date. Internet: `http://asert.arbornetworks.com/2007/05/estonian-ddos-attacks-a-summary-to-date/`, May 2007.

[Naz08] Jose Nazario. Georgia On My Mind - Political DDoS. Internet: `http://asert.arbornetworks.com/2008/07/georgia-on-my-mind-political-ddos/`, July 2008.

[Net07] Netscape Communications Corporation. ODP – Open Directory Project. Online: `http://dmoz.org`, September 2007.

[New04] BBC News. Hacker Threats to Bookies Probed. Internet: `http://news.bbc.co.uk/1/hi/technology/3513849.stm`, February 2004.

[NH08] Jose Nazario and Thorsten Holz. As the Net Churns: Fast-Flux Botnet Observations. In *Proceedings of 3rd International Conference on Malicious and Unwanted Software*, October 2008.

[Nor03] Norman. Sandbox Whitepaper, 2003. Internet: `http://sandbox.norman.no/pdf/03_sandboxwhitepaper.pdf`.

[Nor09] Norman. Norman SandBox Information Center. Internet: `http://sandbox.norman.no/`, Accessed: 2009.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'05)*, 2005.

[OMR08] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. UPX, the Ultimate Packer for eXecutables. Internet: `http://upx.sourceforge.net/`, April 2008.

[Ove07] Claus R. F. Overbeck. Efficient Observation of Botnets. Master's thesis, RWTH Aachen University, May 2007.

[Pat02] David A. Patterson. A Simple Way to Estimate the Cost of Downtime. In *Proceedings of the 16th USENIX System Administration Conference (LISA'02)*, 2002.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. ACM Press, New York, NY, USA, 1988.

[PH07] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, July 2007.

[PMRM08] Niels Provos, Panayiotis Mavrommatis, Moheeb A. Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, 2008.

[PNBH90] Kenneth Hugh Pollock, James D. Nichols, Cavell Brownie, and James E. Hines. *Statistical Inference for Capture-recapture Experiments*. Wildlife Society, 1990.

[PPMB08] E. Passerini, R. Paleari, L. Martignoni, and D. Bruschi. FluXOR: Detecting and Monitoring Fast-Flux Service Networks. In *Proceedings of 5th Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*, pages 186–206, 2008.

[Pro04] Niels Provos. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004.

[PSB06] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[PSY07] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. A Multi-perspective Analysis of the Storm (Peacomm) Worm. Technical report, Computer Science Laboratory, SRI International, October 2007.

[RHW+08] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Proceedings of 5th Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*, pages 108–125, July 2008.

[RIP07] RIPE NCC. DNS Monitoring Services. Internet: `http://dnsmon.ripe.net/dns-servmon/`, Accessed: March 2007.

[RL04] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, pages 159–169, 2004.

[RLM06] Konrad Rieck, Pavel Laskov, and Klaus-Robert Müller. Efficient Algorithms for Similarity Measures over Sequential Data: A Look beyond Kernels. In *Proceedings of 28th DAGM Symposium on Pattern Recognition*, LNCS, pages 374–383, September 2006.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Rya08] Ryan McGeehan et al. GHH - The "Google Hack" Honeypot. Internet: `http://ghh.sourceforge.net/`, Last checked: October 2008.

[RZMT06] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Proceedings of the 6th Internet Measurement Conference*, 2006.

[RZMT07] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. My Botnet Is Bigger Than Yours (Maybe, Better Than Yours): Why Size Estimates Remain Challenging. In *Proceedings of 1st Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, 2007.

[SBEN07] Moritz Steiner, Ernst W. Biersack, and Taoufik En-Najjary. Exploiting KAD: Possible Uses and Misuses. *Computer Communication Review*, 37(5), October 2007.

[Sch00] Bruce Schneier. Inside Risks: Semantic Network Attacks. *Communications of the ACM*, 43(12):168–168, December 2000.

[Sco07] Scott McIntyre. Toxbot Takedown and Provider Paranoia: A Reflection on Modern ISP Incident Response., May 2007. AusCERT Conference.

[SEENB07] Moritz Steiner, Wolfgang Effelsberg, Taoufik En-Najjary, and Ernst W. Biersack. Load Reduction in the KAD Peer-to-Peer System. In *Fifth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'07)*, 2007.

[SENB07] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A Global View of KAD. In *Proceedings of the Internet Measurement Conference (IMC)*, 2007.

[SGD+02] Stefan Saroiu, P. Krishna Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An Analysis of Internet Content Delivery Systems. In *Proceedings of 5th Symposium on Operating System Design and Implementation (OSDI'02)*, 2002.

[Sha09] Shadowserver Foundation. Homepage. Internet: `http://shadowserver.org/wiki/`, Accessed: March 2009.

[SII05] Yoichi Shinoda, Ko Ikai, and Motomu Itoh. Vulnerabilities of Passive Internet Threat Monitors. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[SJB06] Seungwon Shin, Jaeyeon Jung, and Hari Balakrishnan. Malware Prevalence in the KaZaA File-Sharing Network. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 333–338, 2006.

[SKK+97] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a Denial of Service Attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223, May 1997.

[SKK08] Günther Starnberger, Christopher Kruegel, and Engin Kirda. Overbot: A Botnet Protocol Based on Kademlia. In *Proceedings of the 4th Conference on Security and Privacy in Communication Networks (SecureComm'08)*, pages 1–9, 2008.

[SM04] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security & Privacy*, 2(4):46–50, 2004.

[SMPW04] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM'04)*, 2004.

[SMS01] Dug Song, Robert Malan, and Robert Stone. A Snapshot of Global Worm Activity. Technical report, Arbor Networks, November 2001.

[SNDW06] A. Singh, T.W. Ngan, P. Druschel, and DS Wallach. Eclipse Attacks on Overlay Networks: Threats and Defenses. In *Proceedings of Infocom'06*, April 2006.

[Sof] NETSEC Network Security Software. SPECTER Intrusion Detection System. Internet: `http://www.specter.com`.

[Sop06] Sophos. Online Russian blackmail gang jailed for extorting $4M from gambling websites. `http://www.sophos.com/pressoffice/news/articles/2006/10/extort-ddos-blackmail.html`, October 2006.

[SP01] Dawn X. Song and Adrian Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of IEEE Infocom 2001*, April 2001.

[SRR07] Sören Sonnenburg, Gunnar Rätsch, and Konrad Rieck. Large Scale Learning with String Kernels. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 73–103. MIT Press, 2007.

[SS09] Christian Seifert and Ramon Steenson. Capture-HPC Client Honeypot. Internet: `https://projects.honeynet.org/capture-hpc`, Accessed: March 2009.

[ST07] Sandeep Sarat and Andreas Terzis. Measuring the Storm Worm Network. Technical report, Johns Hopkins University, 2007.

[STA01] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of 20th IEEE INFOCOM*, 2001.

[Sta08] StarForce. ASPack – Best Choice Compression and Protection Tools for Software Developers. Internet: `http://www.aspack.com/`, October 2008.

[STC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[Ste] Joe Stewart. Truman - The Reusable Unknown Malware Analysis Net. `http://www.lurhq.com/truman/`.

[Ste07] Joe Stewart. Storm Worm DDoS Attack. Internet: `http://www.secureworks.com/research/threats/storm-worm`, 2007.

[Ste08] Moritz Steiner. *Structure and Algorithms for Peer-to-Peer Cooperation*. PhD thesis, Eurecom and University of Mannheim, December 2008.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities With Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[SWKA00] Stefan Savage, David Wetherall, Anna R. Karlin, and Tom Anderson. Practical Betwork Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, August 2000.

[SWLL06] W. Timothy Strayer, Robert Walsh, Carl Livadas, and David Lapsley. Detecting Botnets with Tight Command and Control. In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, November 2006.

[Sym] Symantec. Decoy Server. Internet: `http://www.symantec.com`.

[Sym06] Symantec. Mantrap. Internet: `http://www.symantec.com/`, Accessed: 2006.

[Tea08] WASTE Development Team. WASTE – An anonymous, secure, and encryped collaboration tool. Internet: `http://waste.sourceforge.net/`, October 2008.

[Tec08] Bitsum Technologies. PECompact2. Internet: `http://www.bitsum.com/pecompact.shtml`, October 2008.

[The03] The Honeynet Project. Know Your Enemy: Sebek, November 2003. `http://www.honeynet.org/papers/sebek.pdf`.

[The05] The Honeynet Project. Know Your Enemy: Phishing, May 2005. `http://www.honeynet.org/papers/phishing/`.

[The07] The Honeynet Project. Know Your Enemy: Fast-Flux Service Networks, July 2007. `http://www.honeynet.org/papers/ff/`.

[Tho08] Thomas P. O'Brien. Two European Men Charged With Conspiring to Launch Cyberattacks Against Websites of Two U.S. Companies. `http://www.usdoj.gov/criminal/cybercrime/walkerIndict.pdf`, October 2008.

[Tri07] Philipp Trinius. Omnivora: Automatisiertes Sammeln von Malware unter Windows. Master's thesis, RWTH Aachen University, Germany, October 2007.

[UML] The user-mode linux kernel home page. Internet: `http://user-mode-linux.sourceforge.net/`.

[Uni07] United States District Court. Indictment: United States of America vs. Lee Graham Walker and Axel Gembe. `http://blog.wired.com/27bstroke6/files/walker_indictment.pdf`, December 2007.

[Vap98] V.N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

[VBBC04] Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure, and Baudouin Le Charlier. The HoneyTank : A Scalable Approach to Collect Malicious Internet Traffic. In *Proceedings of the International Infrastructure Survivability Workshop*, 2004.

[VMC+05] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'05)*, 2005.

[VMw] VMware. Virtual infrastructure software. Internet: `http://www.vmware.com/`.

[Wan09] Kathy Wang. MITRE Honeyclient Development Project. Internet: `http://honeyclient.org`, Accessed: March 2009.

[WBJ+06] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)*, February 2006.

[WC07] Rhiannon Weaver and Michael Collins. Fishing for Phishes: Applying Capture-Recapture Methods to Estimate Phishing Populations. In *Proceedings of 2nd APWG eCrime Researchers Summit*, 2007.

[WFLY] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Short talk presented at CRYPTO'04.

[WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security and Privacy*, 5(2), March 2007.

[Wil06] Carsten Willems. Automatic Behavior Analysis of Malware. Master's thesis, RWTH Aachen University, June 2006.

[YBP04] Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'04)*, 2004.

[YR08] Ting-Fang Yen and Michael K. Reiter. Traffic Aggregation for Malware Detection. In *Proceedings of 5th Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*, 2008.

[ZHH+07] Jianwei Zhuge, Thorsten Holz, Xinhui Han, Chengyu Song, and Wei Zou. Collecting Autonomous Spreading Malware Using High-Interaction Honeypots. In *Proceedings of the 9th International Conference on Information and Communications Security (ICICS'07)*, pages 438–451, December 2007.

[ZHS+08] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying Malicious Websites and the Underground Economy on the Chinese Web . In *Proceedings of 2008 Workshop on the Economics of Information Security (WEIS'08)*, June 2008.