School of Economics and Commercial Law
GÖTEBORG UNIVERSITY
Department of Informatics
2004-05-26

An evaluation of network based sniffer detection; Sentinel

Abstract

Today, tools for sniffer detection have become a standard part of the security toolkit, used to protect computing assets from hostile attacks. The Open Source Network-based sniffer detection tool Sentinel, is commonly found in various security toolkits, and widely used by administrators. Under normal circumstances, Sentinel detects common non-standalone packet sniffers quite reliably. But, its reliability is still questionable. This due to the fact, that since the introduction of Network-based non-standalone sniffer detection, various counter methods have been suggested, to make sniffers impossible to detect. This research effort tries to evaluate the reliability of Network-based sniffer detection, regarding the various counter methods proposed. The research was conducted by standardized experiments conducted with Sentinel, and a survey examination among system administrators. The major findings of this research are that; Network-based sniffer detection, as it is generally conducted today, can not be considered very reliable. Therefore, sniffers should mainly be fought using prevention not detection.

Keywords: Intrusion Detection; Sniffer Detection; Sniffer; Network Security;
Counter Detection;

Author: Daniel Susid
Advisor: Faramarz Agahi
Master of Science Thesis, 20 Swedish Credit Points

Table of Contents

1 Introduction 4
1.1 Research Definition 5
1.2 Research Question 5
1.3 Objective 6
1.4 Scope and limitations 6
1.5 Pictures 6
1.6 Disposition 6
2 Theory 7
2.1 Ethernet Network Interface Cards 7
2.2 TCP/IP on Ethernet 8
2.3 Non-standalone shared Ethernet sniffers 10
2.4 Network based sniffer detection 11
2.4.1 MAC based methods 11
2.4.1.1 ARP detection method 12
2.4.1.2 Etherping detection method 13
2.4.2 Decoy based methods 14
2.4.2.1 DNS detection method 14
2.4.3 Network and machine latency based methods 16
2.4.3.1 Load detection method 16
2.5 Network based sniffer counter detection 18
2.5.1 Modifying the kernel 18
2.5.1.1 Countering the ARP detection method 18
2.5.1.2 Countering the Etherping detection method 20
2.5.2 No DNS lookups 21
2.5.3 Monitoring the network traffic 21
2.5.3.1 Countering load detection 21
3 Method 23
3.1 Research type 23
3.1.1 Experiment setup 23
3.1.2 Experiment execution 24
3.1.2.1 ARP detection experiments 24
3.1.2.2 Etherping detection experiments 24
3.1.2.3 DNS detection experiments 25
3.1.3 Survey examination 25
3.2 Data Collection and Data analysis 26
3.3 Validity and Reliability 26
4 Empirical Results 28
4.1 Experimental results 28
4.1.1 ARP detection experiment results 28
4.1.2 Etherping detection experiment results 29
4.1.3 DNS detection experiment results 29
4.2 Survey examination results 30
5 Discussion and Conclusion 33
5.1 Discussion 33
5.2 Conclusion 35
5.3 Further research 36
6 References 37
6.1 Literature References 37

6.2 Internet References 37
Appendices 39
Appendix 1 – eth.c 40
Appendix 2 – ip_input.c 44
Appendix 3 – asp_lkmachk.c 50
Appendix 4 – fl_aasp.c 52
Appendix 5 – Questionnaire 55

1 Introduction

Since the inception of the Internet various security incidents and attacks have been a continuous phenomenon reaching unparalleled levels, which becomes very evident by reviewing recent studies conducted in the domain by the CC/CERT (2003). These attacks, comprised of different techniques exploiting the Internet in general and TCP/IP in particular, steadily continue to grow in sophistication, using a very large set of tools. Sniffers are counted amongst these tools (AbdelallahElhadj, Khelalfa & Kortebi, 2002; McClure, Scambray, & Kurtz, 2001).

Sniffers were originally developed due to the need for a tool to debug networks. Essentially they capture, interpret and store network data for later analysis. However, just like most powerful tools used by network administrators, sniffers became subverted over the years and are now often used as malicious means to attack various systems (McClure *et al.,* 2001). By using sniffers, malicious users can tap in on the network traffic for information without the knowledge of the networks legitimate owner. This information can include passwords, e-mail messages, encryption keys, sequence numbers or other proprietary data, etc (AbdelallahElhadj *et al.,* 2002; McClure *et al.,* 2001). Often, some of this information can be used to penetrate further into the network, or cause other severe damage. This underlines the importance of reliable sniffer detection that can aid network administrators, in detecting malicious sniffing activities (McClure *et al.,* 2001).

Today, tools for sniffer detection have become a standard part of the security toolkit, used to protect computing assets from hostile attacks (McClure *et al.,* 2001; Mellander, 2001). However, it is important to understand that the installation of a sniffer detector is a second-tier defence. If the sniffer detector detects any unidentified sniffing activities, it means the network has already been penetrated. By receiving and responding to a sniffer detector alert, the intrusion can be limited in scope and halted before further serious damage is incurred. In addition, the sniffer detector alert can aid in computer forensics, and help make attackers more accountable for their actions. Hence, sniffer detectors and Intrusion Detection Systems (IDS) in general, may act as a deterrent to attacks (Mellander, 2001).

It is important to distinguish between stand-alone and non-standalone packet sniffers when it comes to sniffer detection. A stand-alone packet sniffer attached to a local network segment doesn't transmit any packets, and can not be detected by the traditional techniques used in non-standalone sniffer detection. Stand-alone packet sniffers are mainly detected by a combination of using Time Domain Reflectometers (TDR)[1], and physical network inspections. Non-standalone packet sniffers on the other hand, installed as a program on a normal computer, will often generate traffic (Graham, 2000).

Due to its nature, non-standalone sniffer detection is generally divided into two areas (McClure *et al.,* 2001):

- Detection at Local Host Level (Host-based).
- Detection at Local Network Segment Level (Network-based).

Host-based packet sniffer detection is primarily based on examination of the process list, log files and the Network Interface Card (NIC) (McClure *et al.,* 2001). Network-based packet

[1] A TDR is basically RADAR for the wire. It sends a pulse down the wire, and graphs the reflections that come back. An expert can look at the graph of the response and figure out if any devices are attached to the wire that shouldn't be. They also roughly tell where, in terms of distance along the wire, the tap is located.

sniffer detection normally consists of three different types of methods (AbdelallahElhadj *et al.,* 2002; Hawes & Naghibi, 2002):

- MAC based methods.
- Network and machine latency based methods.
- Decoy based methods.

These methods are based on the fact that the non-standalone packet sniffer often generates network traffic, puts a load on the machine used, and that the attacker can be lured with bait traffic to perform detectable actions (AbdelallahElhadj *et al.,* 2002; Hawes & Naghibi, 2002).

The Open Source Network-based sniffer detection tool Sentinel, is commonly found in various security toolkits, and widely used by administrators (McClure *et al.,* 2001). For example, it is included as a standard package in the security toolkit Trinux, a ramdisk-based Linux distribution that boots from a single floppy or CD-ROM (Franz, 2003). Sentinel, which is a consol based application for the Unix/Linux environment, uses two types of methods to remotely detect sniffers (Bind, 2001):

- MAC based methods.
  - ARP detection method.
  - Etherping detection method.
- Decoy based methods.
  - DNS detection method.

Under normal circumstances Sentinel detects common non-standalone packet sniffers quite reliably. But, its reliability is still questionable. This due to the fact, that since the introduction of Network-based non-standalone sniffer detection, the following three counter methods have been suggested within the security community to make sniffers totally passive (Hawes & Naghibi, 2002):

- Modifying the kernel.
- No DNS lookups.
- Monitoring the network traffic.

Theoretically, these counter methods would render a Network-based sniffer detection tool like Sentinel, useless. However, research within this area is limited. Therefore, the feasibility of implementation and effect still needs to be empirically verified.

1.1 Research Definition

This research will analyze and evaluate the Network-based non-standalone sniffer detection tool Sentinel, regarding its reliability under circumstances where counter methods are applied. Furthermore, the research will try to analyze and evaluate the broad opinion regarding the reliability of Network-based detection, among system administrators. The aim of this study is to evaluate the reliability of Network-based sniffer detection in general, regarding the various counter methods proposed.

1.2 Research Question

Given the background described, the following research question has been formulated:

- How reliable are the main Network-based sniffer detection methods used today?

1.3 Objective

Traditionally, research in the area of information and communication security focused on helping developers of systems prevent security vulnerabilities in the systems they produce, before the systems are released to customers. In addition, in most studies on network security, only external attacks were being considered. All of these areas are of the outmost importance when it comes to information and communication security, but need to be complemented with research supporting those developing detection and recovery mechanisms, and studying internal threats.

The purpose of this research is to provide an understanding of Network-based non-standalone sniffer detection, and the reliability of the main detection methods used today.

1.4 Scope and limitations

This paper focuses on the tool Sentinel, and the opinion found among system administrators to determine the reliability of Network-based non-standalone sniffer detection. Sentinel was chosen for this research, due to the fact that it is an Open Source Unix/Linux tool widely used by system administrators, and that it implements two out of three main types of detection methods used today for Network-based non-standalone sniffer detection.

The aim of this research is to determine how the reliability of detection is affected by the following three counter methods; modifying the kernel, no DNS lookups, and monitoring the network traffic. Since Sentinel does not implement any machine and network latency based detection method, it and its corresponding counter method, monitoring the network traffic, has been omitted in the experimental part of this study. However, it is included in the survey examination conducted among various system administrators. Furthermore, due to time constraints, this study has been limited to the Linux platform, and does not cover Network-based sniffer detection in switched Ethernet networks.

In order to comprehend the subject covered in this paper, the reader needs to have a fundamental knowledge of Ethernet networks and the TCP/IP protocol. A basic understanding of Intrusion Detection and the C programming language is also recommended.

1.5 Pictures

The pictures used in this paper all have references, with the exception of the ones created by the author of this paper.

1.6 Disposition

This paper is organized as follows. Section 2 documents the theory behind Network-based non-standalone sniffer detection, and counter detection. The research methods used in this thesis are described in section 3. In section 4, the empirical results from the experiments and survey examination are presented. Finally, conclusions are made and some directions for future research are provided in section 5.

2 Theory

This section is aimed at providing the reader with an understanding of Network-based non-standalone sniffer detection and counter detection. To make the subject at hand more comprehensible to the reader, this section starts of with a brief introduction to NICs, TCP/IP on Ethernet, and non-standalone Ethernet sniffers. Then, in a more profound manner, it explains the different detection and counter detection methods used today.

2.1 Ethernet Network Interface Cards

Each Ethernet NIC comes with a unique Ethernet source address called a Medium Access Control (MAC) address. The MAC address is assigned to the NIC by its manufacturer and is normally stored in a Programmable Read Only Memory (PROM)[2] on the NIC. These addresses are globally unique, and are assigned in blocks of 16 (or 8) million addresses to the Ethernet interface manufacturers, according to a flat addressing structure. This ensures that two Ethernet NICs will never have the same source address. Therefore, all NICs can be uniquely identified by its MAC address (Fairhurst, 2001). Since Ethernet normally is a shared medium, all packets are essentially broadcasted. Due to the inefficiency of passing all the packets broadcasted on the network to the operating system, Ethernet controller chips normally implement a filter which filters out any packet that does not contain a correct destination MAC address for the NIC (Wu & Wong, 1998).

Ethernet NICs can be set to a special state called promiscuous mode. When in promiscuous mode, all Ethernet data packets regardless of the destination MAC address are passed to the operating system (figure 1). Thus, enabling a program running on a machine to set the NIC in promiscuous mode, and capture all the traffic (Wu & Wong, 1998).

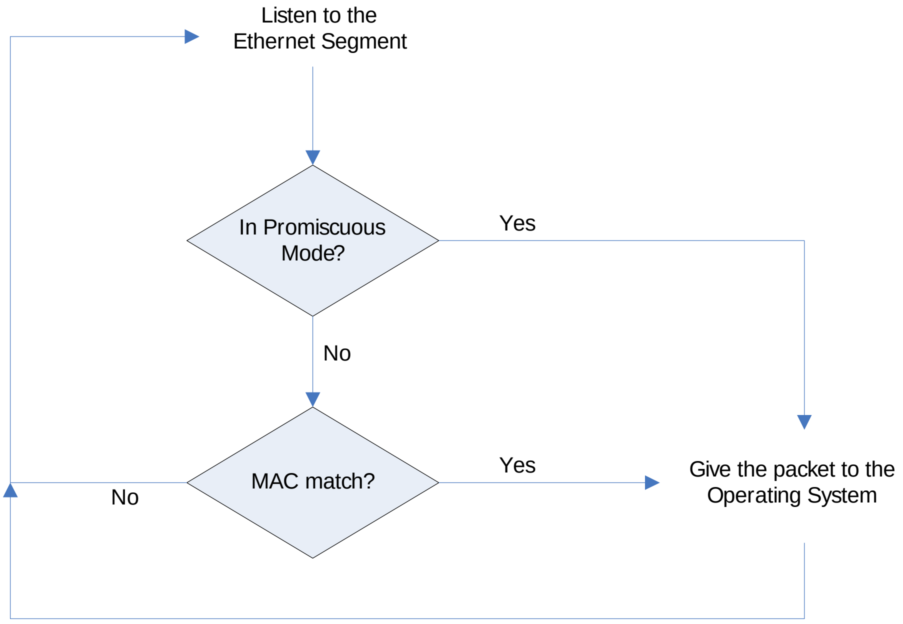


Figure 1: The logic control performed by the NIC (Wu & Wong, 1998).

[2] A PROM is read-only memory (ROM) that can be modified once by a user. PROM is a way of allowing a user to tailor a microcode program using a special machine called a PROM programmer.

2.2 TCP/IP on Ethernet

Normally, networking protocols consist of different layers, with each layer responsible for a different aspect of the communication. The TCP/IP protocol suite is the combination of different protocols at various layers. TCP/IP is normally considered to be comprised of 4-layers, where each layer has its own specific responsibility (figure 2) (Stevens, 2003).

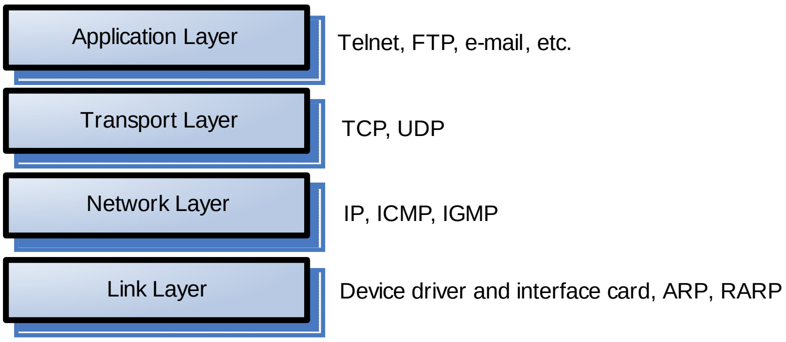


Figure 2: The four layers of the TCP/IP protocol suite (Stevens, 2003).

There are many different protocols in the TCP/IP protocol suite (Stevens, 2003). Figure 3 shows some of them.

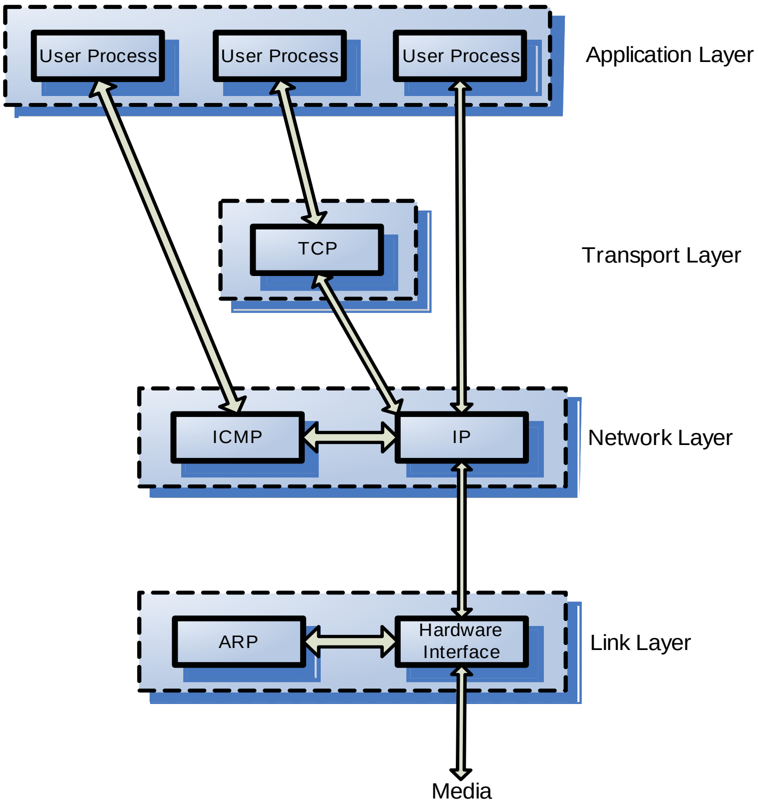


Figure 3: Various protocols at the different layers in the TCP/IP protocol suite (Stevens, 2003).

TCP is one of the two predominant transport layer protocols. It uses IP as the network layer. IP is the main protocol at the network layer. Every piece of data that gets transferred around an internet goes through the IP layer at both end systems and at every intermediate router. ICMP is an adjunct to IP. It is used by the IP layer to exchange error messages and other vital information with the IP layer in another host or router. ARP is a specialized protocol used only with certain types of network interfaces like Ethernet and token ring, to convert between the address used by the IP layer and the address used by the network interface (Stevens, 2003).

When sending data using TCP, the data is sent down the protocol stack of the operating system, passing through each one of its layers, until it is finally sent as a stream of bits across the network. At each layer information is added to the data by prepending headers to the data that is received (figure 4). The unit of data that TCP sends is called a TCP segment. The unit of data that IP sends to the network interface is called an IP datagram. The stream of bits that flows across the Ethernet is called a frame (packet). TCP and ICMP send data to IP that forwards it to the network interface. The network interface sends and receives frames on behalf of IP, ARP, etc (Stevens, 2003).

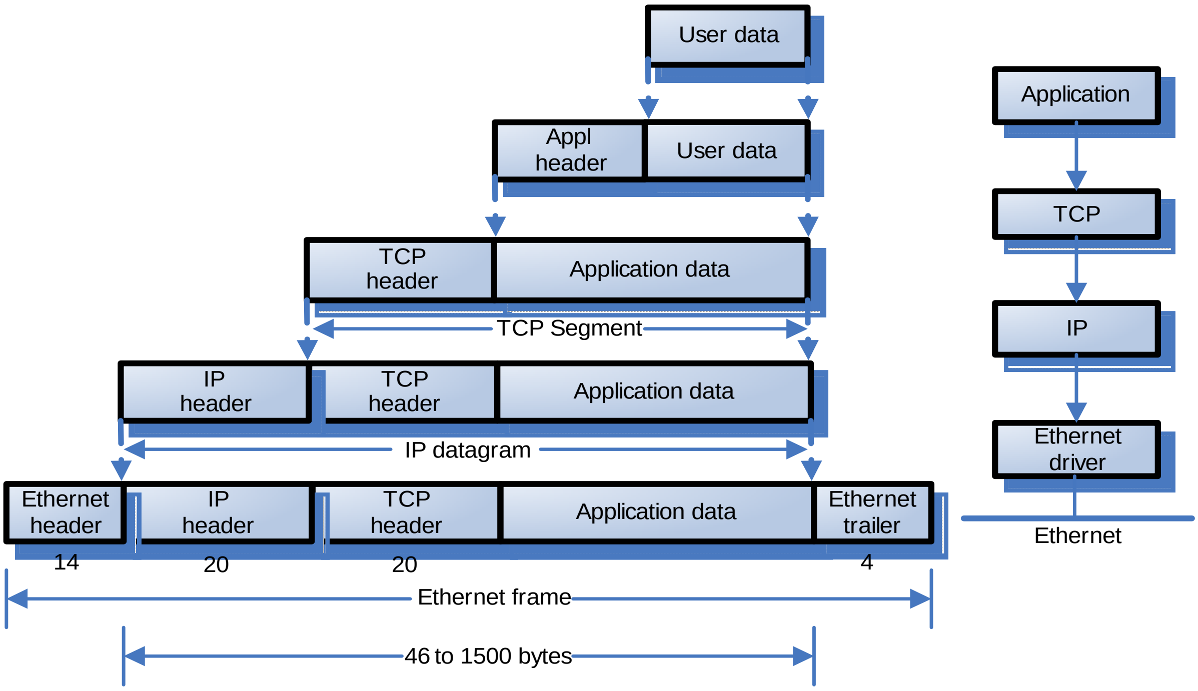


Figure 4: Overview of encapsulation of data as it goes down the protocol stack (Stevens, 2003).

In the world of TCP/IP, the encapsulation of IP and ARP datagrams for Ethernet is defined in RFC 894 (Hornig, 1984). The frame format used uses 48-bit destination and source addresses (figure 5). These 48-bit addresses are the so called MAC addresses. A normal IP packet destined to a particular Ethernet host has the source and destination hosts MAC address filled in the Ethernet header, and the 32-bit IP address of the source and destination filled in the IP header. Thus, IP packets transported by Ethernet have two types of addresses, which normally correspond to the MAC addresses and IP addresses of the source and destination machines. ARP packets transported by Ethernet, also have the source and destination MAC address filled in the Ethernet header. But, the MAC addresses and IP addresses are also filled in the ARP header of the ARP packet (Stevens, 2003).

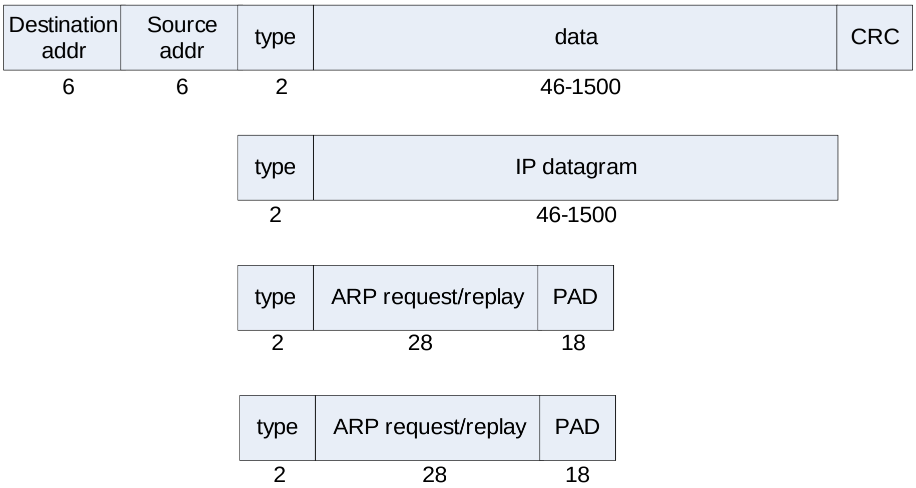


Figure 5: Overview of Ethernet encapsulation (Stevens, 2003).

2.3 Non-standalone shared Ethernet sniffers

A non-standalone shared Ethernet sniffer is software that works coherently with the NIC to capture all traffic within range of the listening system, instead of just the traffic addressed to the sniffing host (McClure *et al.*, 2001). Due to the fact, that shared Ethernet networks are shared communication channels that essentially broadcasts all the packets, the NIC of a computer on these networks has the ability to see all the packets transmitted on the segment it resides on (figure 6) (AbdelallahElhadj *et al.*, 2002).

Every packet that goes through the network contains a header with a MAC address, distinguishing the recipient of the packet. During normal operating procedures, when the NIC is not in promiscuous mode, only the machine with a NIC that has that proper MAC address is supposed to accept the packet, unless the destination MAC address is a broadcast address[3]. Therefore, the NIC must be put in promiscuous mode by the sniffer to enable it to receive all packets floating on the wire (AbdelallahElhadj *et al.*, 2002).

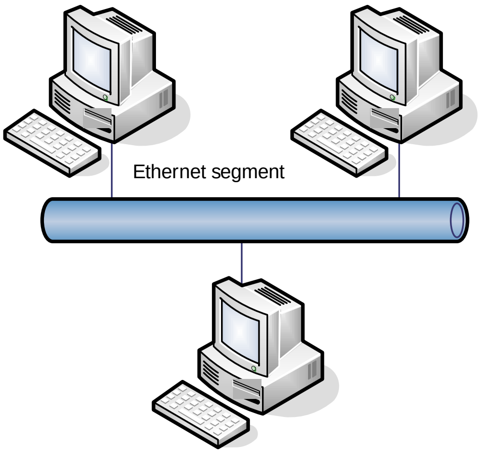


Figure 6: Overview of an Ethernet segment.

[3] A broadcast address is a common address that is used to direct (broadcast) a message to all terminals in a network.

When the NIC is put in promiscuous mode, the sniffer can capture and analyze all the traffic that travels through local Ethernet segment. This of course, means that the range of a sniffer is somewhat limited, because it will not be able to listen to traffic outside of the local networks collision domain. In other words, it can not listen to traffic beyond bridges, routers, switches or other segmenting devices (figure 7). However, if a sniffer is placed on a backbone[4], internetwork link, or other network aggregation point, it will be able to monitor a greater deal of traffic than one placed on an isolated Ethernet segment (McClure *et al.*, 2001).

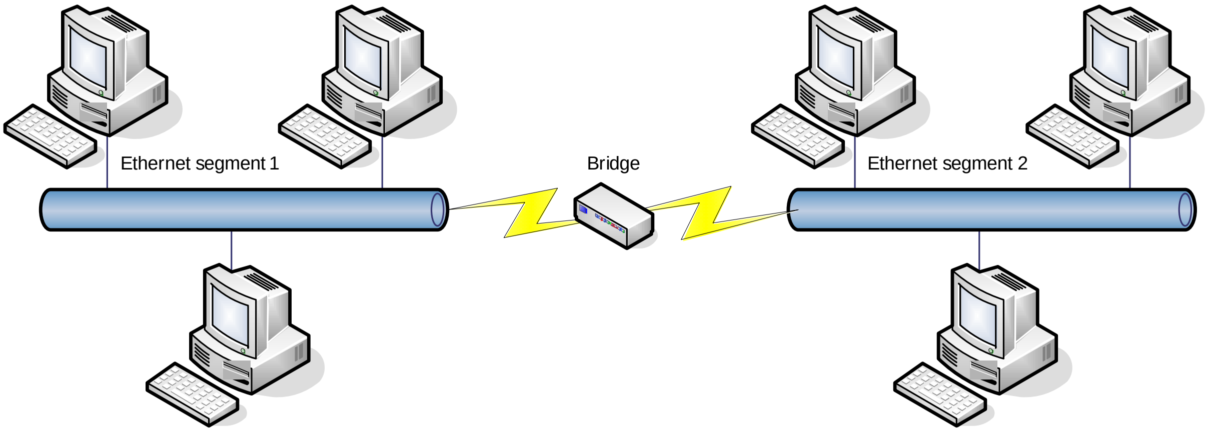


Figure 7: Overview of two Ethernet segments connected by a bridge.

2.4 Network based sniffer detection

The Network-based approach to non-standalone sniffer detection has one very big advantage over host-based detection. It makes it possible to check an entire network from a single point of entry, by using Network-based methods to remotely detect sniffers on a local network segment. By running the sniffer detector from a specific host, the network administrator can perform various tests against other hosts to detect the presence of NICs in promiscuous mode in the network (AbdelallahElhadj *et al.*, 2002). Finding NICs in promiscuous mode, is often a good indication of possible packet sniffing activities (Wu & Wong, 1998).

Detecting sniffers is a daunting task due to their passive nature, which becomes very apparent when reviewing the state of the research in the domain. Very little research has been conducted, and very few Network-based sniffer detectors have been developed during the years (AbdelallahElhadj *et al.*, 2002).

Network-based packet sniffer detection normally consists of three different types of methods; MAC based methods, decoy based methods, and network and machine latency based methods (AbdelallahElhadj *et al.*, 2002).

2.4.1 MAC based methods

The MAC based detection techniques work by exploiting holes found in the implementation of the TCP/IP stack in some operating systems. On some TCP/IP stacks, under certain specific circumstances the destination MAC address of the Ethernet header is never checked or checked insufficiently, when the NIC is in promiscuous mode. Due to this fact, it is

[4] Backbone is another term for bus, the main wire that connects nodes. The term is often used to describe the main network connections composing the Internet.

possible to generate an Ethernet packet with an incorrect MAC address that is passed to the TCP/IP processing code. Normally, such a packet would be rejected by the NIC and therefore never reach the operating system for processing. However, when the NIC is in promiscuous mode, it is actually possible to get these packets processed as if they had a correct MAC address, on some implementations of the TCP/IP stack. The trick for this type of techniques is to elicit a response from the TCP/IP stack, and in such a way determine if an incorrectly addressed packet is acknowledged (AbdelallahElhadj *et al.*, 2002; Wu & Wong, 1998).

Generally, there are two methods based on this technique used today; the ARP detection method and the Etherping detection method (AbdelallahElhadj *et al.*, 2002; Spangler, 2003).

2.4.1.1 ARP detection method

The ARP detection method exploits the flaw in how some operating systems analyze ARP packets. This method uses ARP request packets that are sent to a target with an incorrect MAC address that has the group bit[5] set (figure 8). Normally, such a packet is discarded. But when in promiscuous mode, some operating systems will grab these packets as legitimate packets since the MAC address is checked insufficiently, and respond accordingly. If the target machine replies to the ARP request package with an ARP reply package, we know it is in promiscuous mode. Thus, a sniffer could likely be running on that host (AbdelallahElhadj *et al.*, 2002; Spangler, 2003).

[5] The group bit indicates if a MAC address is an individual address or a group address.

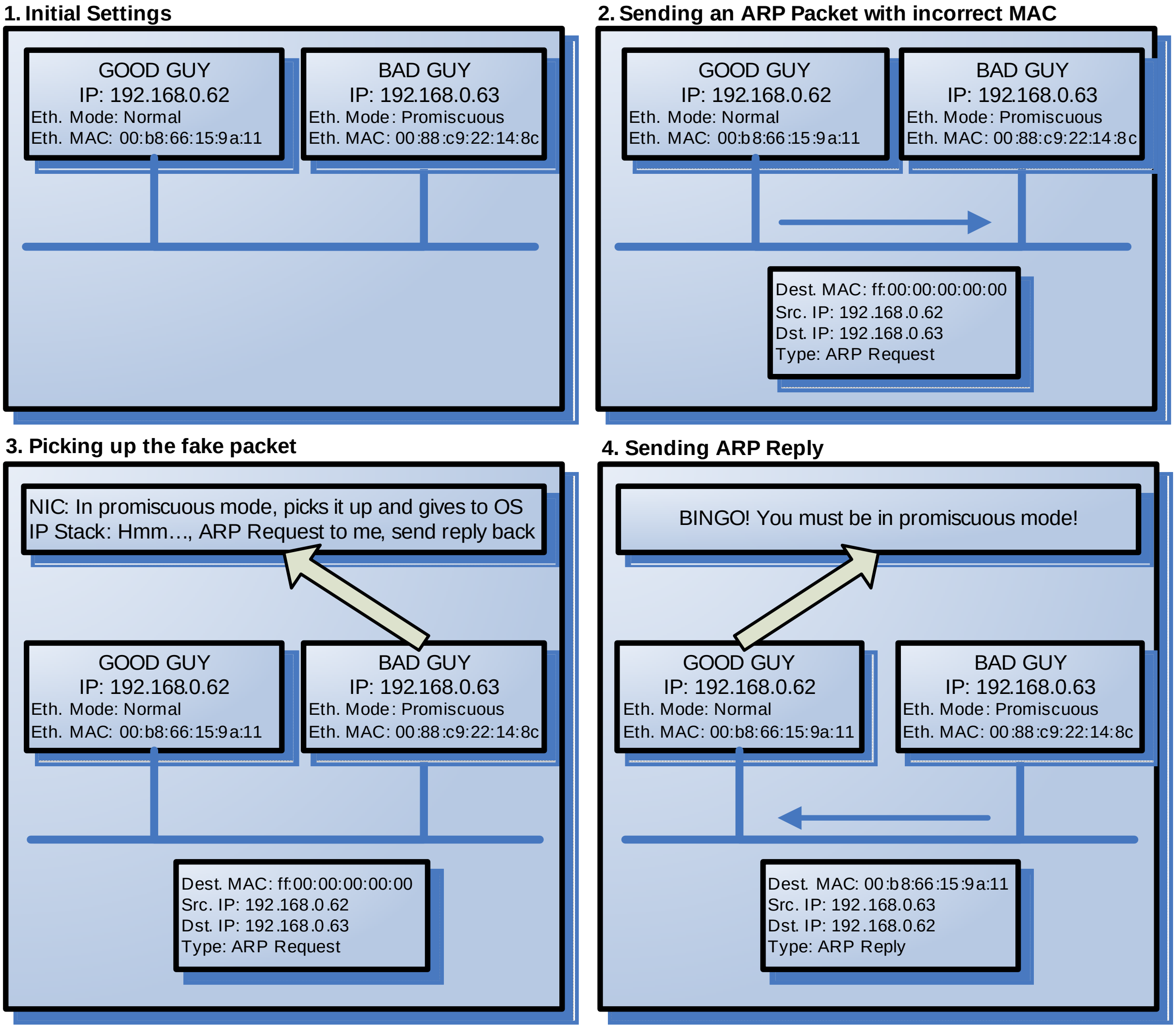


Figure 8: Overview of the MAC based ARP detection method.

2.4.1.2 Etherping detection method

The Etherping detection method exploits the flaw in how many operating systems analyze ICMP packets. This method uses ICMP echo packets that are sent to a target with the correct destination IP address, but with a bogus destination MAC address (fig. 9). Normally, such a packet is discarded. But when in promiscuous mode, some old Linux kernels and NetBSD will grab these packets as legitimate packets since the MAC address is never checked and their IP address is correct, and respond accordingly. If the target in question replies with an ICMP echo reply packet to the request, we know it is in promiscuous mode. Thus, a sniffer could likely be running on that host (AbdelallahElhadj *et al.*, 2002; Wu & Wong, 1998).

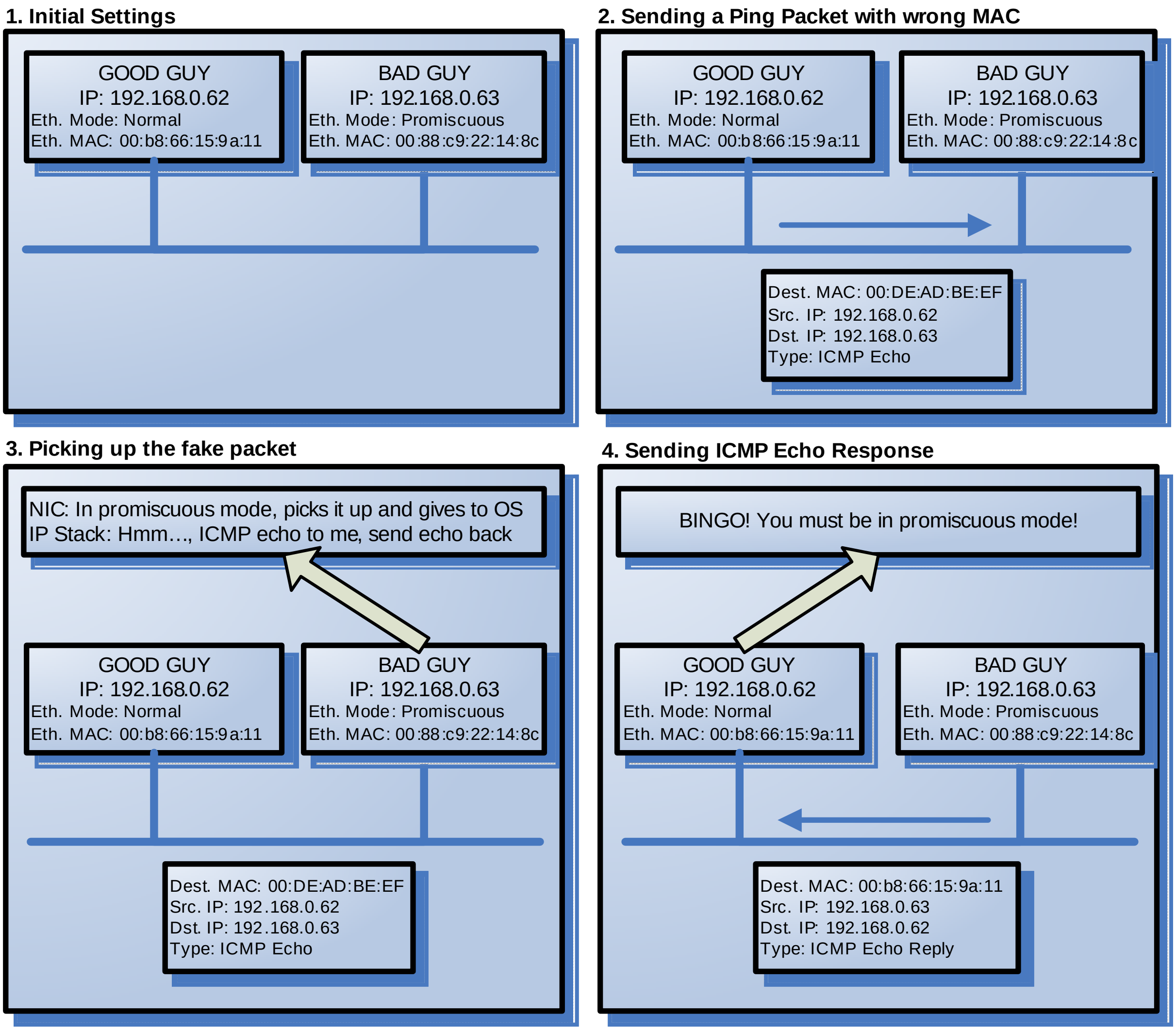


Figure 9: Overview of the MAC based Etherping detection method (Wu & Wong, 1998).

This is an old detection method that no longer is considered reliable. It should only be used for educational purposes (Hawes & Naghibi, 2002).

2.4.2 Decoy based methods

The decoy based detection techniques work on the basis of deceit or honeypot. The idea behind these techniques is to spread out especially attractive bait traffic like false passwords, false user names, fake TCP connections, etc, and await the sniffer owner to launch an attack by reusing the false information. Due to the fact, that nobody except the possible sniffer owner knows the false information, an attack can be distinguished (AbdelallahElhadj *et al.*, 2002).

Generally, one method based on this technique is used today; the DNS detection method (AbdelallahElhadj *et al.*, 2002; Wu & Wong, 1998).

2.4.2.1 DNS detection method

The DNS detection method exploits a behaviour found in most common sniffers. The fact is that most sniffers are not truly passive, but tend to generate traffic, although it is usually hard to distinguish whether the generated network traffic was from the sniffer or not. By default, most sniffers do a reverse DNS lookup on the traffic that it sniffed. Because this traffic is generated by the sniffer program, the trick behind this detection method is to some how detect this DNS lookup, and distinguish it from normal DNS lookup requests (Wu & Wong, 1998).

By generating fake traffic to the Ethernet segment with some unused IP address, it is possible to detect sniffer activity. Since the traffic generated normally should be ignored by the hosts on the segment, if a DNS lookup request is generated, there is a sniffer on the Ethernet segment. One way to implement this type of test is to generate a fake three-way handshake[6] that seems to be a legitimate TCP connection, and then listen to the traffic trying to determine if a DNS lookup is made (figure 10) (AbdelallahElhadj *et al.*, 2002; Wu & Wong, 1998).

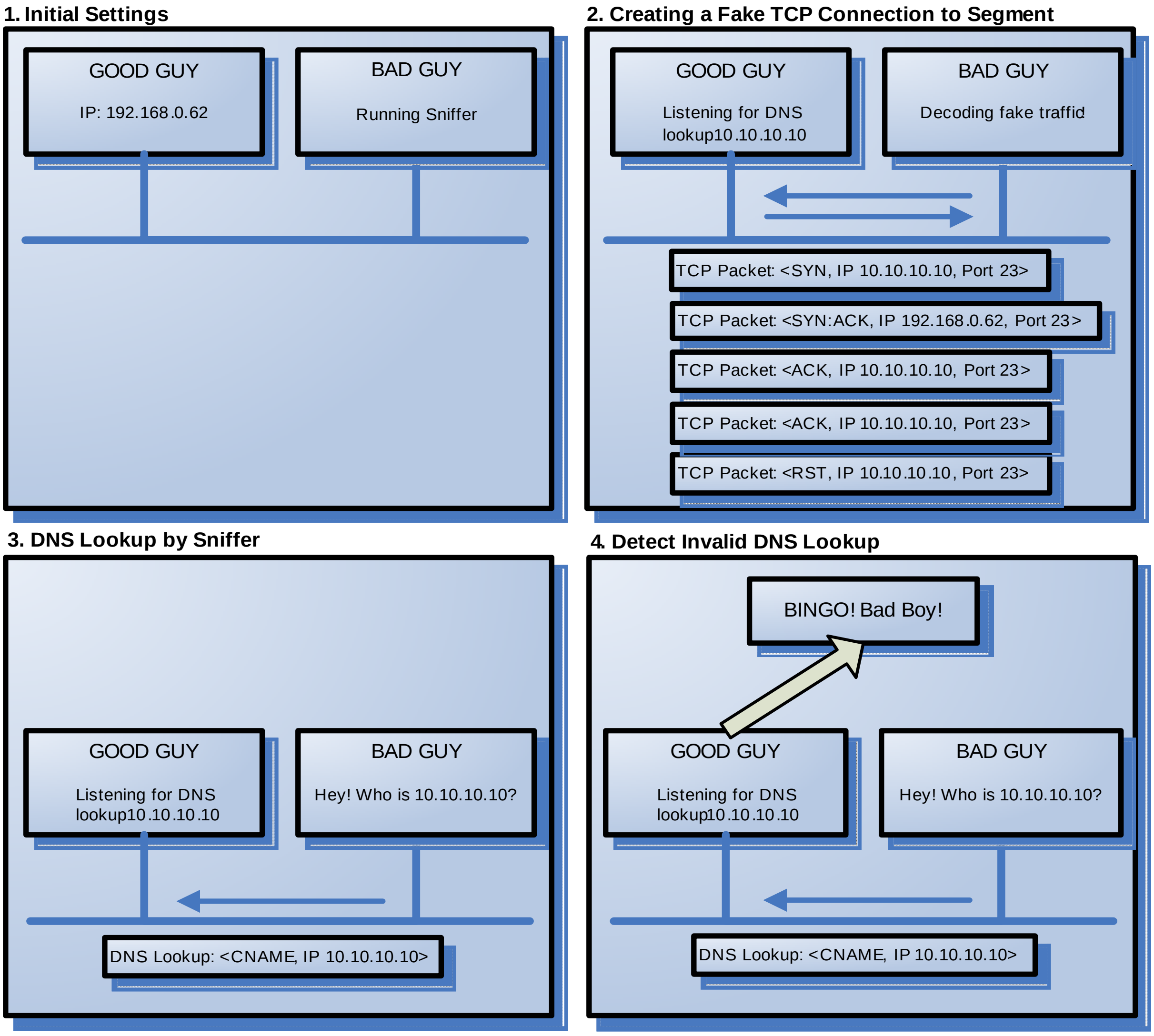

1. Initial Settings
GOOD GUY
IP: 192.168.0.62
BAD GUY
Running Sniffer
2. Creating a Fake TCP Connection to Segment
GOOD GUY
Listening for DNS lookup10.10.10.10
BAD GUY
Decoding fake traffid
TCP Packet: <SYN, IP 10.10.10.10, Port 23>
TCP Packet: <SYN:ACK, IP 192.168.0.62, Port 23>
TCP Packet: <ACK, IP 10.10.10.10, Port 23>
TCP Packet: <ACK, IP 10.10.10.10, Port 23>
TCP Packet: <RST, IP 10.10.10.10, Port 23>
3. DNS Lookup by Sniffer
GOOD GUY
Listening for DNS lookup10.10.10.10
BAD GUY
Hey! Who is 10.10.10.10?
DNS Lookup: <CNAME, IP 10.10.10.10>
4. Detect Invalid DNS Lookup
BINGO! Bad Boy!
GOOD GUY
Listening for DNS lookup10.10.10.10
BAD GUY
Hey! Who is 10.10.10.10?
DNS Lookup: <CNAME, IP 10.10.10.10>

Figure 10: Overview of the Decoy based DNS detection technique.

This technique can be implemented by sending a TCP SYN segment to a unused IP address, followed by a TCP SYN:ACK segment to the earlier segments source IP address, and two TCP ACK segments to the unused IP address. The fake connection finishes of with a TCP RST segment to the unused IP address. Then by listening to the traffic for a DNS query packet for the unused IP address, it is possible to determine if there is a sniffer on a specific host (Bind, 2001).

[6] The three-way handshake is the TCP connection establishment protocol.

2.4.3 Network and machine latency based methods

The network and machine latency based detection techniques work by noting degrading system performance, caused by NIC interrupts to the operating system. When sniffers are running, they put the NIC in promiscuous mode as mentioned earlier in this paper. When in promiscuous mode, all Ethernet traffic passing the NIC will generate hardware interrupts which in turn will cause the Ethernet driver code to execute. Furthermore, all the captured packets must be passed to the user program running the sniffer. It is a known fact that crossing the kernel boundary is quite expensive. Therefore, under heavy traffic, a sniffer can heavily degrade performance on a host with a NIC in promiscuous mode (AbdelallahElhadj *et al.,* 2002; Wu & Wong, 1998).

The trick for this type of techniques is to somehow remotely measure the machine load when there is heavy traffic on the network segment. This can be accomplished by taking a measurement of response time from the machine that is suspected of running a sniffer. However, how this measurement is taken is often the most difficult part in implementing this type of technique (AbdelallahElhadj *et al.,* 2002; Wu & Wong, 1998).

One method based on this technique, is the load detection method described by Hawes and Naghibi (2002).

2.4.3.1 Load detection method

In the load detection method, two measurements of response time are taken (figure 11). One measurement is taken to determine the response time of the machine without heavy network traffic, and the other measurement is taken to determine the response time of the machine with heavy traffic. The load detection method is based on the assumption that the sniffer does some parsing. A very large amount of ICMP request packets with an unused destination address is sent on the network flooding it. Meanwhile, a computer which is suspected to be running a sniffer has been sent an ICMP echo request packet before, and during the flooding stage. The machine will parse the data if it is in promiscuous mode, which increases the load on it. Extra time is needed for this increased load, so it will take longer to respond to the ICMP echo request packet with an ICMP echo reply packet. The difference in the response times of the suspected machine, and other machines indicates that the suspected machine is in promiscuous mode. In which case, a sniffer could likely be running on that host (Hawes & Naghibi, 2002).

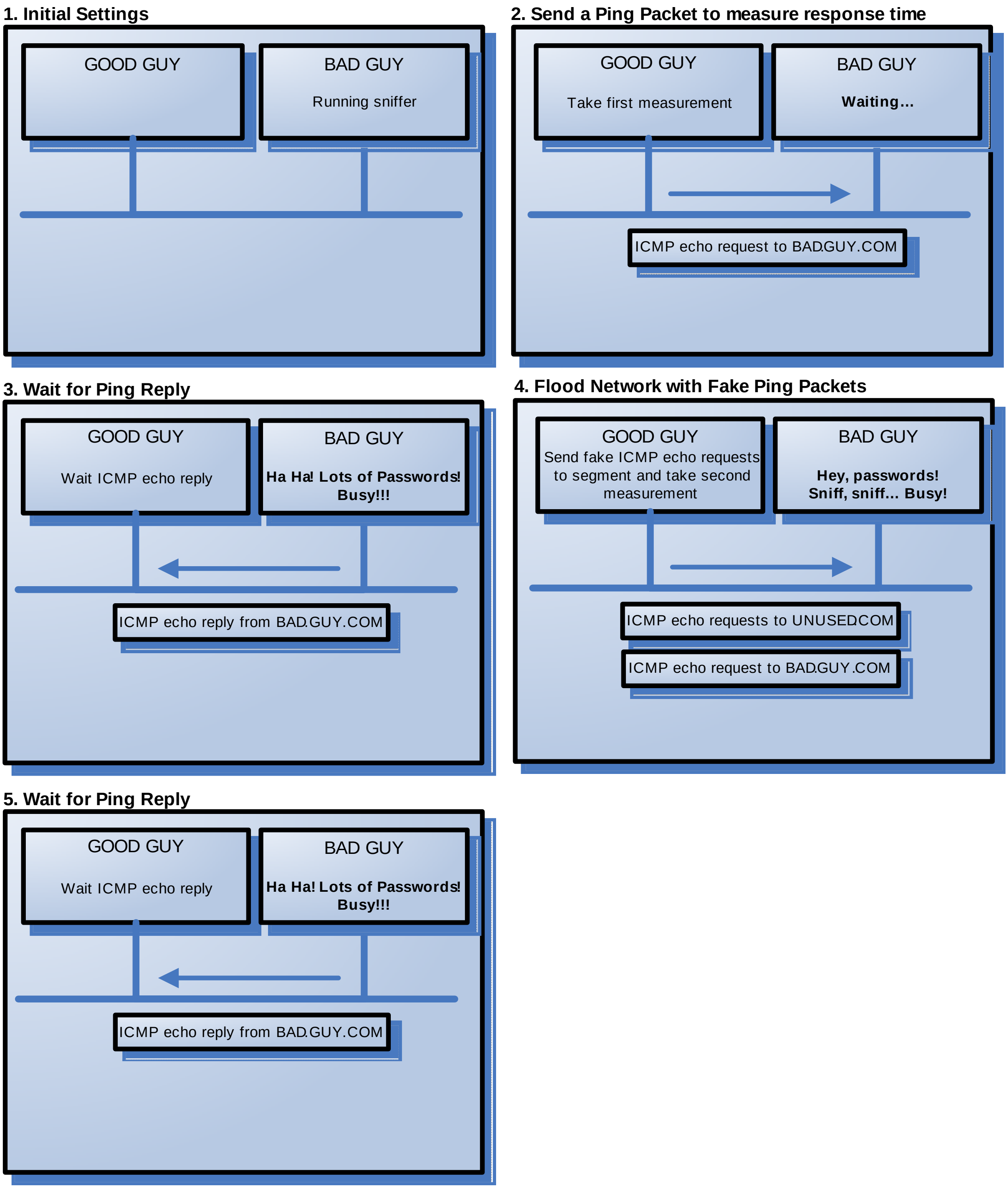

1. Initial Settings
GOOD GUY
BAD GUY
Running sniffer
2. Send a Ping Packet to measure response time
GOOD GUY
Take first measurement
BAD GUY
Waiting…
ICMP echo request to BADGUY.COM
3. Wait for Ping Reply
GOOD GUY
Wait ICMP echo reply
BAD GUY
Ha Ha! Lots of Passwords!
Busy!!!
ICMP echo reply from BAD.GUY.COM
4. Flood Network with Fake Ping Packets
GOOD GUY
Send fake ICMP echo requests to segment and take second measurement
BAD GUY
Hey, passwords!
Sniff, sniff… Busy!
ICMP echo requests to UNUSED.COM
ICMP echo request to BADGUY.COM
5. Wait for Ping Reply
GOOD GUY
Wait ICMP echo reply
BAD GUY
Ha Ha! Lots of Passwords!
Busy!!!
ICMP echo reply from BAD.GUY.COM

Figure 11: Overview of the network and machine latency based load detection technique.

The main problem with this type of technique is that it can degrade the overall network performance. Furthermore, it is susceptible to timeouts and false positives, due to the fact that packets can be delayed simply because of the load on the network. Another problem is the possibility of false positives due to the fact that many sniffers are user mode programs, while ICMP echo requests are responded to in kernel mode, which means that they are independent of the CPU load on the machine (AbdelallahElhadj *et al.*, 2002).

2.5 Network based sniffer counter detection

Since the introduction of Network-based non-standalone sniffer detection, various counter methods have been suggested within the security community to make sniffers totally passive. These counter detection methods theoretically pose a serious problem, to the reliability of Network-based detection (Hawes & Naghibi, 2002).

Generally, the counter detection methods discussed are; modifying the kernel, no DNS lookups, and monitoring the network traffic (Hawes & Naghibi, 2002).

2.5.1 Modifying the kernel

The kernel modification counter-detection technique, works by altering the kernel networking code of the operating system. This is done to render MAC based detection techniques, useless. The MAC based techniques are dependent on implementation holes in the TCP/IP stack that make it feasible to determine, if a machine is in promiscuous mode. By altering the kernel code and in such making exploitation of these holes impossible, a non-standalone sniffer running on such an altered system would be impossible to detect, using MAC based techniques (Hawes & Naghibi, 2002).

Since Linux is an open source operating system, it is possible to examine its networking code to know how different packets are processed (Linux, 2004). This makes it quite simple to implement counter detection methods for the MAC based techniques.

2.5.1.1 Countering the ARP detection method

After ARP packets bypass the NIC, they are first received by the Ethernet module and then passed on to the ARP module. In the Ethernet module, the first thing that is checked is the MAC address group bit. If the group bit is set the MAC address is classified as `PACKET_BROADCAST` if it matches the broadcast address FF:FF:FF:FF:FF:FF, otherwise it is classified as `PACKET_MULTICAST` by the Ethernet module. However, if the group bit is not set, the MAC address is classified either as `PACKET_OTHERHOST` if it does not match the local MAC address or to us if it does (AbdelallahElhadj *et al.*, 2002).

The hole that the ARP technique utilizes lies at this level, and is based on the fact that when the group bit is set and the MAC address does not match the broadcast address, it is automatically classified as multicast. There is never any check done to validate that the MAC address is a legitimate multicast address. This can be seen by reviewing the following code, found in function eth_type_trans() in the Ethernet module (appendix 1) (AbdelallahElhadj *et al.*, 2002):

```
/* Check the group bit of the destination MAC address */
if(*eth->h_dest&1)
{
            /* Check if MAC matches broadcast address FF:FF:FF:FF:FF:FF */
            /* else classify as multicast */
            if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
                        skb->pkt_type=PACKET_BROADCAST;
            else
                        skb->pkt_type=PACKET_MULTICAST;
}

/*
 *          This ALLMULTI check should be redundant by 1.4
 *          so don't forget to remove it.
 *
 *          Seems, you forgot to remove it. All silly devices
 *          seems to set IFF_PROMISC.
 */

else if(1 /*dev->flags&IFF_PROMISC*/)
{
            /* Check if destination MAC is that of the NIC otherwise */
            /* classify as other host */
            if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
                        skb->pkt_type=PACKET_OTHERHOST;
}
```

By changing the code stated above, to perform a more proper check of the destination MAC address. The hole that the ARP technique utilizes should be patched, and the technique rendered useless. This can be achieved by the following changes to the code stated above:

```
/* Check the group bit of the destination MAC address */
if(*eth->h_dest&1)
{
            /* Check if MAC matches broadcast address FF:FF:FF:FF:FF:FF */
            /* else classify as other host */
            if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
                        skb->pkt_type=PACKET_BROADCAST;
            else
                        skb->pkt_type=PACKET_OTHERHOST;
}

/*
 *          This ALLMULTI check should be redundant by 1.4
 *          so don't forget to remove it.
 *
 *          Seems, you forgot to remove it. All silly devices
 *          seems to set IFF_PROMISC.
 */

else if(1 /*dev->flags&IFF_PROMISC*/)
{
            /* Check if destination MAC is that of the NIC otherwise */
            /* classify as other host */
            if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
                        skb->pkt_type=PACKET_OTHERHOST;
}
```

The modification made, changes the check of the destination MAC address when the group bit is set, to classify all non broadcast addresses as `PACKET_OTHERHOST` instead of `PACKET_MULTICAST`. This means that any packet with a destination MAC not that of the NIC and not the broadcast address, will be rejected.

The change shown above is only meant as a quick fix to achieve the wanted experimental results. For a permanent solution to the problem, a much better approach would probably be to check the MAC address if classified as multicast, against a list of valid multicast addresses. Furthermore, the above change is done statically to the kernel code, which makes it necessary to recompile the kernel, and reboot the machine to achieve the wanted results (Bovet & Cesati, 2003). Out of a malicious user perspective, this would probably pose a problem. This due to the fact, that a kernel recompilation and reboot, makes it very difficult to preserve the stealth most likely desired. Therefore, a much better way to modify the kernel, than the static way, is to use dynamically Loadable Kernel Modules (LKM)[7] (Bovet & Cesati, 2003). By using LKMs, the malicious user could very easily patch the hole exploited by the ARP detection method, without any recompilation or reboot.

[7] LKMs are used by the Linux kernel to expand its functionality. The advantage of LKMs: They can be loaded dynamically; there must be no recompilation of the whole kernel. Because of these features, they are often used for specific device drivers (or file systems) such as soundcards etc.

The following code found in the chk_mac_arp() function in the asp_lkmachk LKM (appendix 3) written by a person calling himself Vecna (2000), checks the destination MAC addresses of ARP packets to see if they match the broadcast address or the local device (to us):

```
/* checks if the destination MAC is that of the broadcast address 0xffffffffffff */
if( r_mac[0] ==r_mac[1] ==r_mac[2] ==r_mac[3] ==r_mac[4] ==r_mac[5] ==0xff)
            /* mac broadcast */
            goto end;

/* checks if the destination MAC is that of the local device */
if( (r_mac[0] !=t_mac[0]) || (r_mac[1] !=t_mac[1]) ||
    (r_mac[2] !=t_mac[2]) || (r_mac[3] !=t_mac[3]) ||
    (r_mac[4] !=t_mac[4]) || (r_mac[5] !=t_mac[5]) )
{
            /* ARP mac spoof detected */
            sk->nh.arph->ar_hrd = 0;
            sk->nh.arph->ar_pro = 0;
            sk->nh.arph->ar_op = 0;
            goto end;
}
```

By dynamically linking the asp_lkmachk LKM to the kernel, the above stated code would render the ARP detection method useless in the same manner as the static modification shown earlier.

2.5.1.2 Countering the Etherping detection method

When ICMP packets bypass the NIC, they are first received by the Ethernet module and then passed on to the IP module, which in turn forwards them to the ICMP module (Welte, 2000). In earlier Linux kernel versions, the MAC address was not checked which made it possible to send ICMP echo request packets with a wrong MAC address, and still get a reply when the NIC was in promiscuous mode (Wu & Wong, 1998). But in newer versions of the Linux kernel, the MAC address is checked, which can be seen by reviewing the following code found in function ip_rcv() in the IP module (appendix 2):

```
/* When the interface is in promisc. mode, drop all the crap
 * that it receives, do not try to analyse it.
 */
if (skb->pkt_type == PACKET_OTHERHOST)
            goto drop;
```

The code stated above, shows that any IP packet, ICMP or not, will be rejected if the destination MAC address has been classified as `PACKET_OTHERHOST` in the Ethernet module. Therefore, there is no need to try and counter the Etherping detection technique when using newer kernels, due to the fact that the exploitation hole has already been patched. However, if using older kernels, one could patch the hole by statically adding a check like the one shown above, or using the earlier mentioned asp_lkmachk LKM (appendix 3) by Vecna (2000). The following code found in the function check_mac_ip() in the asp_lkmachk LKM, checks the MAC address of IP packets:

```
/* checks if the destination MAC is that of the broadcast address 0xffffffffffff */
if( r_mac[0] ==r_mac[1] ==r_mac[2] ==r_mac[3] ==r_mac[4] ==r_mac[5] ==0xff)
            /* mac broadcast*/
            goto end;

/* checks if the destination MAC is that of the local device */
if( (r_mac[0] !=t_mac[0]) || (r_mac[1] !=t_mac[1]) ||
    (r_mac[2] !=t_mac[2]) || (r_mac[3] !=t_mac[3]) ||
    (r_mac[4] !=t_mac[4]) || (r_mac[5] !=t_mac[5]) )
{
            /* IP check - anti spoof detect! */
            sk->nh.iph->tot_len = 0;
            sk->nh.iph->check = 0;
            goto end;
}
```

By dynamically linking the asp_lkmachk LKM to an older kernel, the above stated code would render the Etherping detection method useless in the same manner as the check performed in newer kernels.

2.5.2 No DNS lookups

The no DNS lookups counter-detection technique, works by avoiding so called DNS lookups (Hawes & Naghibi, 2002). The DNS, or Domain Name System, is a distributed database that is used by TCP/IP applications to map between hostnames and IP addresses. From a sniffer applications point of view, access to the DNS is through a resolver. On Unix/Linux hosts the resolver is accessed primarily through two C standard library functions, gethostbyname() and gethostbyaddr(), which are linked with the application when the application is built. The first takes a hostname and returns an IP address, and the second takes an IP address and looks up a hostname. The resolver contacts one or more name servers to do the mapping (Stevens, 2003).

Due to the fact that the resolver is normally part of the application and not the kernel as are the TCP/IP protocols (Stevens, 2003). There is no need to do any kernel modification to prevent DNS lookups. Instead, this can be achieved at the application level by only using IP addresses and avoiding the DNS resolver. This is very easily done, and most common sniffers have the built in option to not use the DNS resolver (Hawes & Naghibi, 2002). For example, when using the common non-standalone sniffer tcpdump, one can easily turn of the usage of the DNS resolver by passing it the argument –n, e.g. (Jacobson, Leres, & McCanne, 2003):

```
[root@linuxbox] tcpdump -n
```

The same can be accomplished when using the common non-standalone sniffer ethereal (Combs, 2004):

```
[root@linuxbox] tethereal –n
```

2.5.3 Monitoring the network traffic

The counter-detection techniques based on monitoring the network traffic are used to counter the network and machine latency based detection methods (Hawes & Naghibi, 2002). Network and machine latency based methods, try to somehow remotely measure the machine load when there is heavy traffic on the network segment. This is done by taking a measurement of response time, from the machine that is suspected of running a sniffer (AbdelallahElhadj *et al.*, 2002; Wu & Wong, 1998). By manipulating this response time, it is possible to counter this type of detection method (Vecna, 2000).

2.5.3.1 Countering load detection

As stated earlier in this thesis, the load detection method is based on two measurements of response time. One measurement is taken to determine the response time of the machine without heavy network traffic, and the other measurement is taken to determine the response time of the machine with heavy traffic. The measurements are implemented by sending an ICMP echo request packet, and waiting for an ICMP echo reply packet to determine the response time. To achieve the heavy traffic, the network is flooded with ICMP echo request packets to an unused address (Hawes & Naghibi, 2002).

Under normal circumstances, the response time depends on the following factors; the network velocity, the network driver, the machine CPU, the network programs at the raw level, the amount of traffic in the network, and the amount of traffic directed at the specific machine in the network. When flooding the network with traffic to an unused address, the machines with a NIC in promiscuous mode will have an increased CPU and network load, which in turn will lead to a longer response time. By using a program that runs in the background monitoring the network traffic, and replying to ICMP echo requests with ICMP echo replies, independently of the kernel, it is possible to decrease the response time. The response time can be decreased in this manner, to the extent of rendering the load detection technique based on ICMP packets useless (Vecna, 2000).

Vecna (2000) provides and example (appendix 4), showing how a program that counters the load detection based on ICMP packets can be implemented. The example uses a special library called libvsk. Libvsk is a library for network traffic manipulation from userlevel, with some functions for filtering and sniffing (Libvsk, 2000).

3 Method

This research uses qualitative and quantitative methods for gathering, processing and analyzing the data. The gathering of empirical data is conducted by qualitative standardized experiments, and a quantitative survey examination. Processing of the acquired data is done by codifying the outcomes of the experiments and the survey examination into easily understood and separable groups, which are then analyzed.

Inference will be drawn from this information by deduction. The research hypothesis, which is based on theoretical assumptions regarding the reliability of Sentinel, will be tested against the actual outcomes of the experiments, and the survey examination resulting in a hypothetic-deductive study.

First, the various standpoints taken when setting up the experiments are described and explained. Next, the procedure of executing the experiments is described. Then, the various standpoints taken when conducting the survey examination are detailed. And last the validity and reliability of the experiments, and the survey examination is discussed.

3.1 Research type

According to Alvager and Beach (1992), there are two main methods in which information can be obtained from an investigated system in scientific and technological research, the observational method and the experimental method. The observational method involves taking records in a passive way. The researcher's study the phenomena as it is presented, taking notes and trying to formulate laws from the observed facts. In the experimental method, the researchers can create new situations and study the results without relying on conditions given by nature. An experiment can be defined as, the acquisition of data to measure the performance of the solution under controlled conditions in a laboratory.

The research type selected for this study is the experimental method, which is suitable since the purpose is to gather empirical data regarding the selected phenomena under controlled conditions. However, to increase the research validity, the experimental method was complemented with a survey examination.

3.1.1 Experiment setup

The experiments where conducted in a small functional shared Ethernet Network, consisting of three hosts connected through a hub (figure 12). The simple Ethernet network configuration was setup at the thesis author's home, without any supervision, using two old Intel Pentium II 300MHz computers, each with 256MB RAM and a Netgear FA311 10/100 PCI NIC, and one Intel Pentium IIII 2,6GHz computer with 512MB RAM and an Intel EtherExpress Pro 10/100Mbit PCI NIC. The three computers were connected using a Netgear DS104 10/100Mbit hub and three Unshielded Twisted Pair (UTP)[8] Cat.5 Ethernet network cables.

Host A and host B, comprised of the two old Intel Pentium 300MHz computers in the network, where acting designated attack hosts, and where therefore running the two common non-standalone sniffers tcpdump and ethereal (figure 12). Host C on the other hand,

[8] An UTP cable is a popular type of cable used in computer networking that consists of two shielded wires twisted around each other. Typically, UTP is used in environments like an office, where there is not that much heavy noise adjacent to the wire that might cause interference.

comprised of the Intel Pentium IIII 2,6GHz computer, was used for detection purposes, and was therefore running Sentinel (figure 12). The operating system used on all the hosts, was Gentoo Linux.

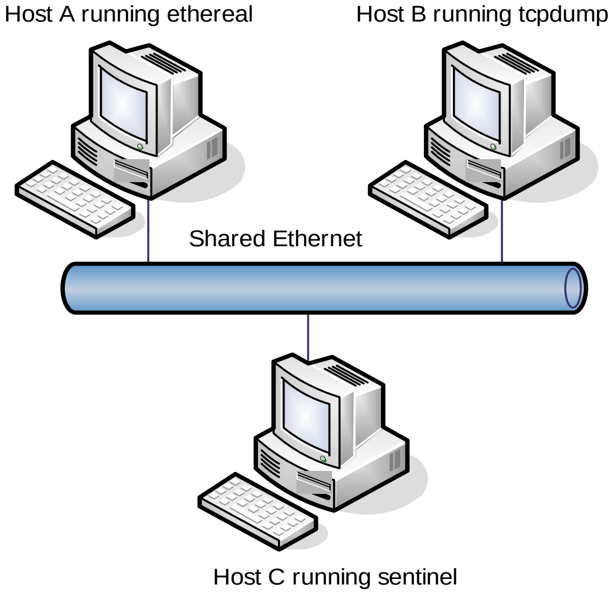


Figure 12: Overview of the experiment network configuration.

3.1.2 Experiment execution

The different experiments executed in this research, can be divided into three main groups; ARP detection experiments, Etherping detection experiments and DNS detection experiments.

3.1.2.1 ARP detection experiments

The ARP detection experiments were executed by running the two common non-standalone sniffers tcpdump and ethereal, on two different hosts (figure 12). Sentinel, installed on a third separate host, was then used to try and detect the two sniffers, using its implemented ARP detection method.

To test the ARP method properly, two experiments were executed. The first experiment was aimed at testing the method under normal circumstances, to assure its and the experimental setups functionality. The second experiment was aimed at testing the method under special circumstances, where a counter method was applied. Therefore, both hosts running the sniffers where using a normal kernel during the first experiment, and a modified kernel during the second experiment. The kernel was modified according to the ARP counter detection technique, described earlier in this thesis.

All ARP detection experiments where conducted on both kernel version 2.4.25 and 2.6.4

3.1.2.2 Etherping detection experiments

The Etherping detection experiments were executed by running the two common non-standalone sniffers tcpdump and ethereal, on two different hosts (figure 12). Both hosts where running a normal kernel. Sentinel, installed on a third separate host, was then used to try and detect the two sniffers, using its implemented Etherping detection method.

Due to the fact that the Etherping method is an old technique, exploiting a hole that should be patched in newer Linux kernel versions. It should not be necessary to modify the kernel, to counter this technique.

All Etherping detection experiments where conducted on both kernel version 2.4.25 and 2.6.4.

3.1.2.3 DNS detection experiments

The DNS detection experiment was executed by running the two common non-standalone sniffers tcpdump and ethereal, on two different hosts (figure 12). Sentinel, installed on a third separate host, was then used to try and detect the two sniffers, using its implemented DNS detection method.

To test the DNS method properly, two experiments were executed. The first experiment was aimed at testing the method under normal circumstances, to assure its and the experimental setups functionality. The second experiment was aimed at testing the method under special circumstances, where a counter method was applied. Therefore, during the second experiment, the sniffers where passed the argument for no DNS lookups.

3.1.3 Survey examination

The survey examination conducted in this thesis is based on the definition by Denscombe (2000) of the self administrated questionnaire, which normally is sent out by regular mail. In this study however, the questionnaire was sent out by electronic mail to the various system administrators taking part. To achieve the desired purpose, the questionnaire was designed to collect information about Network-based non-standalone sniffer detection in general. The questionnaire consisted of a series of uncomplicated firm questions, which were identical for all recipients (appendix 5). This approach was chosen, due to the possibility of reducing the amount of information to the area of interest (Halvorsen, 1992). Furthermore, it simplified the possibility of asking a great number of people the same questions. The reason for using firm instead of open questions was that the firm questions tend to be more clear and precise, and make it easier to compare answers from different respondents.

Due to time constraints, the survey did not reach out to the great number of system administrators that was first intended, which unfortunately resulted in a lack of randomness. Instead, it was conducted among twenty system administrators, which were selected on the basis of practical experience and knowledge of Linux and network security. These system administrators were employed at different universities and companies, with an IT-department. No emphasize was put on the size and type of the companies and universities. The competence of these system administrators was assured by the fact that the author of this thesis came in contact with most of them, through the focus-ids mailing list. The focus-ids mailing list is comprised of a large community of very dedicated people with a very strong interest in the domain, many of them being seasoned professionals, belonging to the top echelon in the area of IDS. However, a few of the system administrators were not approached through the focus-ids mailing list, but where friends of the author, selected due to the authors high regard of their knowledge and experience of the domain.

Despite the lack of randomness, the conducted survey should still give a good indication of the general perception among system administrators. But, under different circumstances, a greater number of participants in the survey would be preferred.

Due to the sensitive nature of information and communication security, the system administrators agreed to participate in the survey, only on condition of anonymity. Weaknesses found in IT-infrastructure, can cause severe damage to the reputation and success of companies and universities. It is a widely known phenomena that people with malicious intent, search through any possible source of information to find possible weaknesses in the IT-infrastructure of a possible target. This of course, includes surveys conducted in the domain. Therefore, the names of the participants, universities and companies, are not mentioned in this paper.

3.2 Data Collection and Data analysis

There are two main ways of gathering and making sense of data used to derive knowledge in research, qualitative and quantitative methods (Thurén, 1991).

Qualitative methods have a low degree of formalization. These methods are primarily used as a mean to provide understanding of a specific phenomenon. The purpose is not necessarily to test whether the information holds in general. The central theme is to get a deeper comprehension of the problem context of study (Thurén, 1991).

Quantitative methods are more formalized and structured. These methods are to a larger extent characterized by control from the researcher. Design and planning using these methods are characterized by selectivity and distance to the information source. This is necessary to conduct a formalized analysis and comparison, and to test whether the result achieved hold in a more general sense. Statistical measurement methods play a central role during the analysis of quantitative information (Thurén, 1991).

This research is based on both qualitative and quantitative methods for data gathering and data analysis. By using this approach, it is possible to attain the qualitative and quantitative aspects of the phenomenon covered in this thesis. The qualitative method is used to get a deeper comprehension of the problem covered in this study, and the quantitative method to test whether the results achieved hold.

The empirical results from the experiments are analyzed in a qualitative manner, and the empirical results from the survey examination are analyzed in a quantitative manner, to provide an answer to the research question.

3.3 Validity and Reliability

When conducting research, two very important things are the reliability and validity (Thurén, 1991). Andersen (1994) states that the evaluation of validity and reliability relies on an exact formulated and adequate research problem, and that the definitions used within are precise, adequate, and free from ambiguity.

Validity refers to what extent you investigate what you intend to investigate. To clarify this, any conclusions regarding the outcome must be based on well founded assumptions, regarding the empirical data’s relationship to the problem or hypothesis. It is often a question of judgment and the goal of the research, whether the validity of the research is high or not (Halvorsen, 1992).

Reliability describes the likelihood that the same results would come out of independent studies, under the same prerequisites. If the likelihood is high, then the reliability is also high (Halvorsen, 1992).

The research question stated in this research and the definitions used, have been carefully formulated regarding precision, adequacy, and with clarity in mind. Furthermore, the methods used in this research, and the analysis are explained to the extent that the results should be replicable by other researchers. In order to provide a high degree of validity in this research, the definition of what is to be investigated, and the methods are thoroughly explained. Since there have been minimal previous research within the subject area of this thesis, it is hard to estimate the reliability. I am convinced though, that a similar study under the same prerequisites, would have given the same results. A way to increase the reliability would be to; perform similar studies with other Network-based non-standalone sniffer detection software on various other platforms, conduct an experimental evaluation of the network and machine latency based methods and their corresponding counter methods, and having a larger amount of participating system administrators in the survey examination. Due to the time limitations for a master thesis, this was not reasonable.

4 Empirical Results

This research comprised of an experimental evaluation of the Network-based sniffer detection tool Sentinel, and a survey examination of Network-based sniffer detection in general, conducted among various system administrators. In this section, the results of the latter will be accounted for.

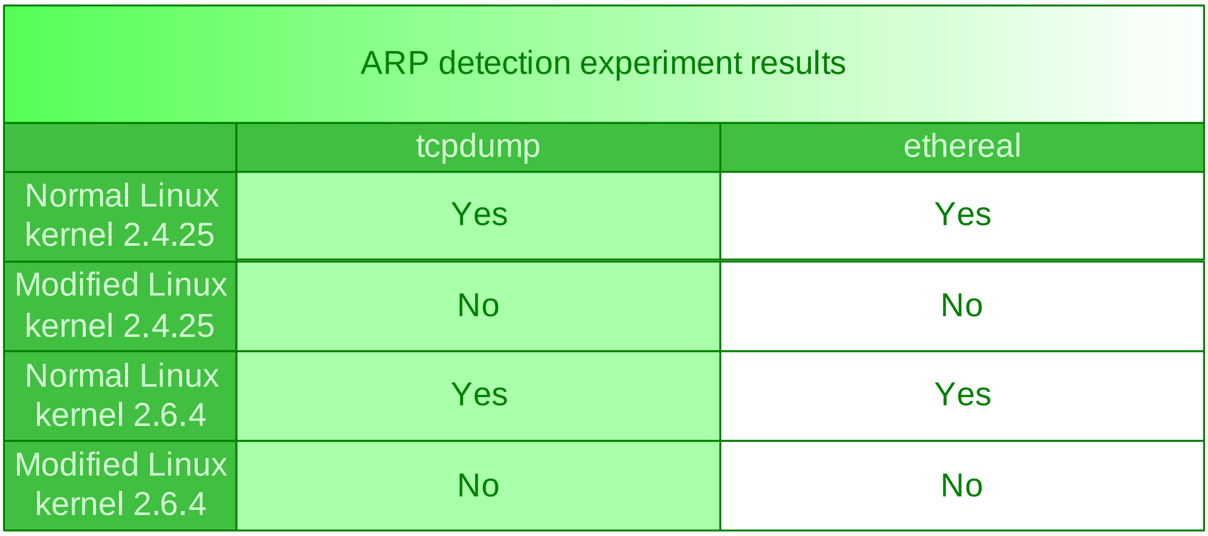
4.1 Experimental results

The experimental results achieved in this study, consist of three different types of experiments; ARP detection experiments, Etherping experiments, and DNS detection experiments. These experiments where conducted using the sniffer detection tool Sentinel, and the two common non-standalone sniffers tcpdump and ethereal, as described earlier in this paper.

4.1.1 ARP detection experiment results

During the ARP detection experiment the two sniffers tcpdump and ethereal, were running in default mode on their respective designated attack hosts. Sentinel running on another host, was then used to try and detect the two sniffers.

After completing the ARP detection experiments, the following results where achieved (figure 13):



ARP detection experiment results		
	tcpdump	ethereal
Normal Linux kernel 2.4.25	Yes	Yes
Modified Linux kernel 2.4.25	No	No
Normal Linux kernel 2.6.4	Yes	Yes
Modified Linux kernel 2.6.4	No	No

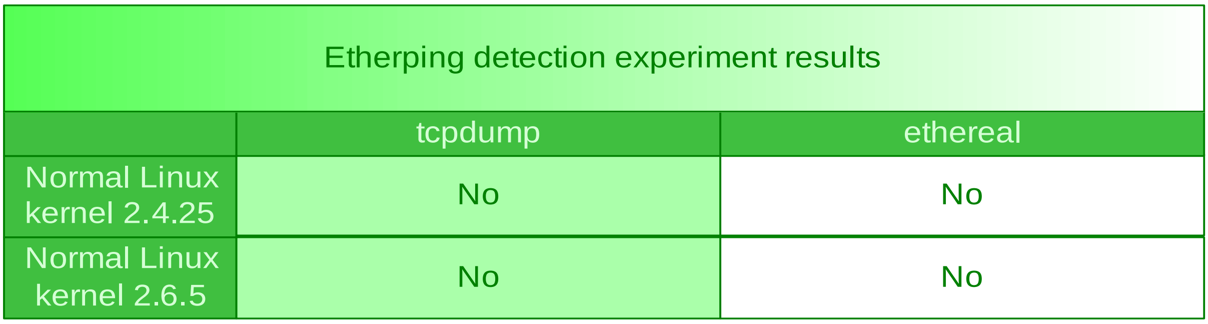
Figure 13: Overview of the results achieved in the ARP detection experiments.

Under normal conditions running both Linux kernel 2.4.25 and 2.6.5, tcpdump and ethereal were detected by Sentinel with its implemented ARP detection method. However, when running the modified versions of the Linux kernel 2.4.25 and 2.6.4, neither tcpdump or ethereal were detected by Sentinel with the ARP detection method.

4.1.2 Etherping detection experiment results

During the Etherping experiment the two sniffers tcpdump and ethereal, were again running in default mode on their respective designated attack hosts. Sentinel running on another host, was then used to try and detect the two sniffers.

After completing the Etherping detection experiments, the following results where achieved (figure 14):



Etherping detection experiment results

	tcpdump	ethereal
Normal Linux kernel 2.4.25	No	No
Normal Linux kernel 2.6.5	No	No

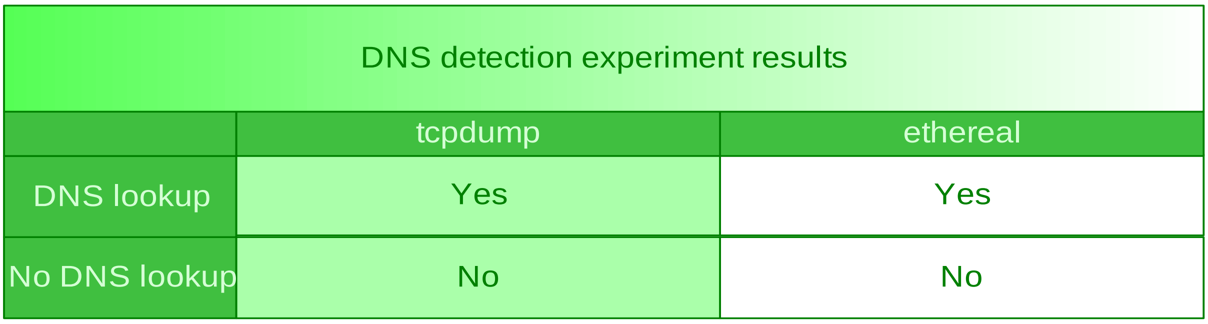
Figure 14: Overview of the results achieved in the Etherping detection experiments.

Under normal conditions running both Linux kernel 2.4.25 and 2.6.5, tcpdump and ethereal were never detected by Sentinel with its implemented Etherping detection method. There was no need to conduct this experiment with a modified kernel.

4.1.3 DNS detection experiment results

During the DNS experiment, the two sniffers tcpdump and ethereal were running in default mode on their respective designated attack hosts in the first experiment, and with the no DNS lookup argument passed to them in the second experiment. Sentinel running on another host was then used to try and detect the two sniffers.

After completing the DNS detection experiments, the following results where achieved (figure 15):



DNS detection experiment results

	tcpdump	ethereal
DNS lookup	Yes	Yes
No DNS lookup	No	No

Figure 15: Overview of the results achieved in the DNS detection experiments.

When running tcpdump and ethereal in default DNS lookup mode, both were detected by Sentinel with its implemented DNS detection method. However, when tcpdump and ethereal where passed the argument for no DNS lookups, neither was detected by Sentinel with the DNS detection method.

4.2 Survey examination results

The first survey examination question (appendix 5), asking if the administrators perceived Network-based sniffer detection reliable, gave the following results (figure 16):

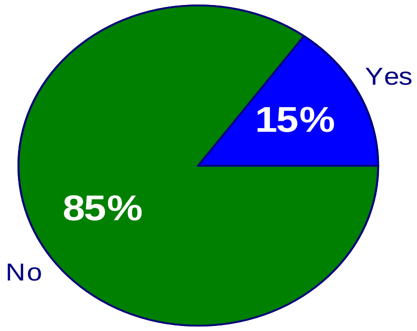


Figure 16: Overview of the results achieved in reply to the first survey examination question; Do you find Network-based non-standalone sniffer detection reliable?

A very large majority, 85% to be exact, clearly found Network-based sniffer detection unreliable. Only 15% found it reliable.

The second survey examination question (appendix 5), asking if the administrators perceived counter detection methods a threat to the reliability of Network-based sniffer detection, gave the following results (figure 17):

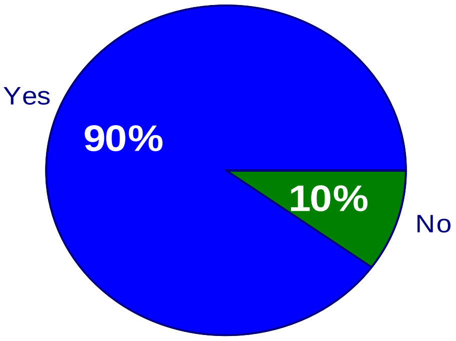


Figure 17: Overview of the results achieved in reply to the second survey examination question; Do you think counter detection methods pose a threat to the reliability of Network-based non-standalone sniffer detection?

As many as 90% of the system administrators, participating in the survey examination, found that counter detection methods pose a threat to the reliability of Network-based sniffer detection. The remaining 10% did not perceive the counter detection methods as a threat.

The third survey examination question (appendix 5), asking if the system administrators perceived the very nature of Network-based sniffer detection, as an obstacle to the development of reliable detection methods, gave the following results (figure 18):

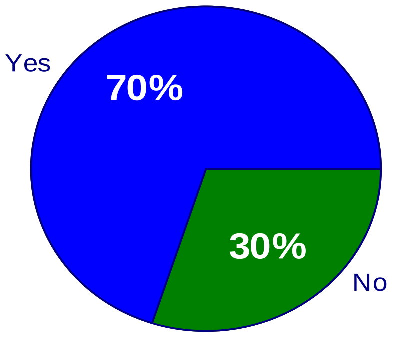


Figure 18: Overview of the results achieved in reply to the third survey examination question; Do you think that the very nature of Network-based non-standalone sniffer detection makes it unfeasible to develop reliable detection methods?

70%, a clear majority of the system administrators, participating in the survey examination, found the very nature of Network-based sniffer detection as an obstacle to the development of reliable detection methods. The remaining 30% did not perceive it as a problem.

The fourth survey examination question (appendix 5), asking if the system administrators thought that the Open Source nature of Linux, affects the feasibility of implementation of counter detection methods against Network-based non-standalone sniffer detection, gave the following results (figure 19):

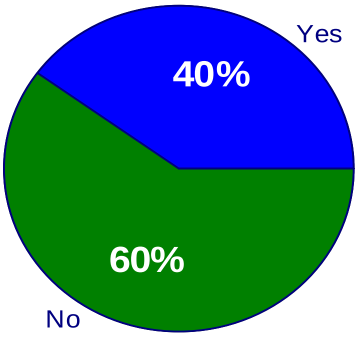


Figure 19: Overview of the results achieved in reply to the fourth survey examination question; Do you think that the Open Source nature of Linux, affects the feasibility of implementation of counter detection methods against Network-based non-standalone sniffer detection?

Only 40% of the system administrators found that the Open Source nature of Linux affects the feasibility of implementation of counter detection methods. The remaining 60% did not think it had any affect on the feasibility.

The fifth and last survey examination question (appendix 5), asking which Network-based sniffer detection method the system administrators perceived as most reliable, gave the following results (figure 20):

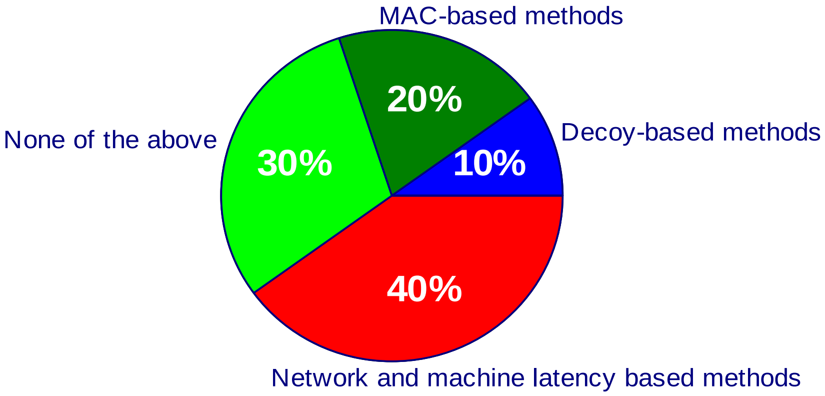


Figure 20: Overview of the results achieved in reply to the fifth survey examination question; Which type of Network-based non-standalone sniffer detection method do you find most reliable?

It is very evident, from the results achieved, that the system administrators, deemed the network and machine latency based methods most reliable. As many as 40% chose network and machine latency based methods. Decoy-based methods on the other hand, were found not to be among the most reliable types of methods, with only 10% choosing them. MAC-based methods also achieved a very low percentage, with only 10% choosing them as the most reliable. As many as 30%, chose none of the above.

5 Discussion and Conclusion

This final chapter concludes this research by a discussion, and brief conclusion of the results achieved in the empirical analysis of Network-based non-standalone sniffer detection. Finally, further research in this application domain is recommended.

5.1 Discussion

Network-based non-standalone sniffer detection is an extremely difficult task, due to the passive nature of sniffers. The fact that counter detection methods exist, that can make sniffers more passive or even totally passive, is not very encouraging. In most cases it makes Network-based sniffer detection, an unreliable tool for administrators that does not fulfil its intended goal.

When reviewing the experimental results achieved in this study, it becomes very evident that the main methods used today for Network-based sniffer detection, do not perform well under circumstances where counter methods are applied. In fact, they are rendered useless. All of the detection methods used in Sentinel; ARP detection method, Etherping detection method, and DNS detection method, failed to detect the two common sniffers tcpdump and ethereal, when the counter detection methods were applied. This fact, together with the ease with which these counter detection methods can be implemented by skilful attackers, strengthens the theory that sniffers should mainly be fought using prevention, not detection. This theory was also further strengthened by the opinion found among various administrators. According to the survey examination results, the majority of system administrators taking part in the study, perceived Network-based non-standalone sniffer detection as unreliable. Furthermore, the majority also found counter detection methods to be a threat to the reliability of Network-based sniffer detection. Therefore, preventative means like encryption, active hubbs, one time passwords, and non-promiscuous NICs should be strongly encouraged. By using these means of prevention, any packet sniffing would be rendered useless or even impossible, depending on the choice of solution (Figure 21).

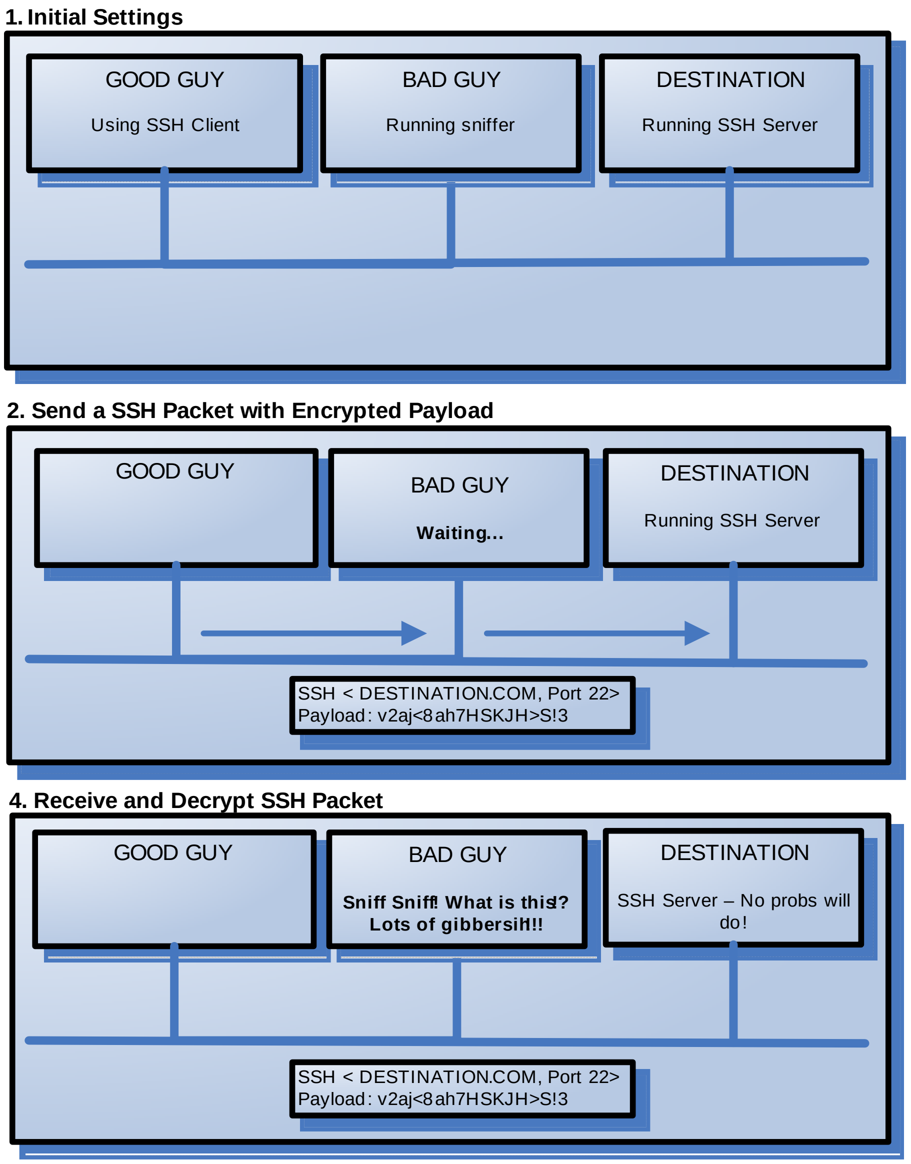

1. Initial Settings
GOOD GUY
Using SSH Client
BAD GUY
Running sniffer
DESTINATION
Running SSH Server
2. Send a SSH Packet with Encrypted Payload
GOOD GUY
BAD GUY
Waiting...
DESTINATION
Running SSH Server
SSH < DESTINATION.COM, Port 22>
Payload: v2aj<8ah7HSKJH>S!3
4. Receive and Decrypt SSH Packet
GOOD GUY
BAD GUY
Sniff Sniff! What is this? Lots of gibberish!!
DESTINATION
SSH Server – No probs will do!
SSH < DESTINATION.COM, Port 22>
Payload: v2aj<8ah7HSKJH>S!3

Figure 21: Overview of sniffer prevention using encrypted traffic with SSH (Secure Shell).

Besides preventative means, another possibility would be to use Host-based detection which is primarily based on examination of the process list, log files and the NIC. This is done using standard system or specially crafted tools, developed with the intention of detecting sniffers on the specific host. Even though Host-based detection would appear to be somewhat more reliable than Network-based detection, it is by no means immune to manipulation and counter measures. For example, tools can be trojanized, and the kernel can be modified to hide processes and the fact that the NIC is in promiscuous mode. With other words, it is important to understand that preventative solutions are most probably the only ones that really provide any real security.

Even though the counter detection methods used in this study, proved easy to implement and very effective, these findings need to be empirically verified against other platforms than Linux. The chance exists that the Open Source nature of Linux, makes the implementation of some of them much easier than it would be in closed source proprietary operating systems like Windows. This again, is based on the assumption that closed source software is harder to exploit, which a lot of people within the security community would not agree on. Generally, the opinion found among security professionals, is that there is no security by obscurity. All software can be reverse engineered, and some even find binaries to be better than source code, when it comes to exploitation. Furthermore, the survey examination results seem to strengthen the opinion that obscurity should not be to large an obstacle. A majority of the system administrator, taking part in the survey, found that the Open Source nature of Linux did not affect the feasibility of implementation of counter detection methods. However, even if obscurity does not cause any difficulties, the effectiveness of the counter detection methods is not guaranteed on other platforms, but theoretically the same principals should apply.

Another interesting aspect worth considering is that the high availability of common sniffers is a very important factor, to why sniffers are such a popular attack tool. However, custom made sniffers that implement counter detection methods to apply stealth, with the exception of the no DNS lookup argument, are not that common. Furthermore, to implement the counter detection methods shown in this study, the malicious user would need to have some fundamental networking and programming skills, which probably reduces the number of potential attackers using these methods. But, the counter detection methods still pose a very realistic threat, and make the reliability of Network-based sniffer detection very questionable, which becomes very apparent after reviewing the results achieved in this study.

The machine and latency based detection methods were never experimentally evaluated in this study, due to the fact that Sentinel does not implement any such methods. But, the results achieved in the survey examination, showed a rather large amount of system administrators considering this type of methods to be more reliable than the other main types of detection methods used today. This of course, makes it possible to believe that implementing this type of method could increase the reliability, of a Network-based sniffer detection tool like Sentinel. However, methods exist to counter this type of methods too, like the one described in this paper, which somewhat undermines this idea. Another problem with this type of methods is that they can degrade network performance, and could therefore be problematic to use in real life networks that depend on good performance. A solution to this problem could be to run this type of tests during the night, when nobody is using the network. But still, it does not seem like machine and network latency based detection methods provide any real solution, to the reliability problem of Network-based sniffer detection.

The very nature of Network-based sniffer detection; exploiting holes in the TCP/IP protocol, measuring the load on the suspected machine, luring the attacker with bait traffic, seems to be constituted of vague factors and assumptions, and therefore appears to have no real possibility of assuring a high degree of reliability, as it is conducted today. Under controlled conditions, these factors and assumptions seem to hold, but at the slightest change problems occur, which is very evident after reviewing the experimental evaluation conducted in this thesis. This phenomena makes it quite easy to develop counter methods, which in turn leads to the fact that as soon as new detection methods are developed, based on the same prerequisites, counter methods are sure to follow. A good indication of the problematic nature of Network-based sniffer detection is that since its introduction, very few Network-based sniffer detection tools have been developed. Furthermore, the survey examination shows that the system administrators, taking part in the study, are of the same opinion. A clear majority found that the nature of the Network-based sniffer detection makes it unfeasible to develop reliable detection methods.

5.2 Conclusion

Concluding this research is best done by answering the research question; How reliable are the main Network-based sniffer detection methods used today?

The major findings of this research are that; by using the proposed counter methods it was possible to effectively counter all the detection methods implemented in Sentinel, which seriously inflicts doubt to whether Network-based detection can be considered reliable. Furthermore, the conducted survey examination seems to strengthen this theory. So in other words, it is very evident by reviewing the experimental and survey results, that the answer to the research question is that Network-based sniffer detection, as it is generally conducted today, can not be considered very reliable. This research clearly supports the theory that the main Network-based detection methods used today are not sufficient to provide reliable sniffer detection, and that sniffers should mainly be fought using prevention not detection.

5.3 Further research

Further research in the area of sniffer detection and counter detection, could include; an experimental evaluation of network and machine latency based detection countering, the feasibility of implementation of counter detection methods on other platforms, and the applicability of counter detection methods for IDS hiding.

Network and machine latency based detection methods, while not used in Sentinel, are implemented in other sniffer detection tools. A need exists to conduct an experimental analysis and evaluation of this type of detection techniques, under circumstances where counter-detection methods are applied. Furthermore, the feasibility of implementation of the different counter-detection methods on other platforms than Linux, also need empirical verification and should be researched. Another very interesting area, that also should be researched, is the applicability of the different counter-detection methods for IDS hiding.

6 References

6.1 Literature References

AbdelallahElhadj, H., Khelalfa, H., & Kortebi, H. (2002). *An Experimental Sniffer Detector: SnifferWall.* Basic Software Laboratory, CERIST.

Alvager, T.K.E. & Beach, D.P. (1992). *Handbook for Scientific and Technical Research.* Prentice-Hall.

Andersen, H. (1994). *Vetenskapsteori och Metodlära En Introduktion.* Studentlitteratur.

Bovet, Daniel., & Cesati, Marco. (2003). *Understanding the Linux Kernel.* O'reilly.

Halvorsen, Knut. (1992). *Samhällsvetenskaplig metod.* Studentlitteratur.

Hawes, Christopher., & Naghibi, Farzaneh. (2002). *CSIC 471 Security Project: Detecting Sniffers.* St. Francis Xavier University.

Hornig, C. (1984). *"Standard for the Transmission of IP Datagrams over Ethernet Networks"*. RFC 894.

Martyn, Denscombe. (2000). *Forskningshandboken.* Studentlitteratur.

McClure, Stuart., Scambray, Joel., & Kurtz, George. (2001). *Hacking Exposed.* McGraw-Hill.

Spangler, Ryan. (2003). *Packet Sniffer Detection with AntiSniff.* University of Wisconsin, Department of Computer and Network Administration.

Stevens, Richard. (2003). *TCP/IP Illustrated, Volume 1: The Protocols.* Addison Wesley.

Thurén, T. (1991). *Vetenskapsteori för nybörjare.* Liber.

Wu, David., & Wong, Frederick. (1998). *Remote Sniffer Detection.* Computer Science Division, University of California, Berkley.

6.2 Internet References

Bind. (2001). *The sentinel project.* [online document 2004-03-17]. URL http://www.packetfactory.net/Projects/sentinel

CERT/CC. (2003). *Module 2 – Internet Security Overview.* [online document 2004-03-01]. URL http://www.cert.org/present/cert-overview-trends/module-2.pdf

Combs, Gerald. (2004). *The Ethereal man page.* [online document 2004-03-29]. URL http://www.zevils.com/cgi-bin/man/man2html?ethereal+1

Fairhurst, Gorry. (2001). *Network Interface Card.* [online document 2004-06-01]. URL http://www.erg.abdn.ac.uk/users/gorry/course/lan-pages/nic.html

Franz, Matthew. (2003). *Trinux: Linux Security Toolkit.* [online document 2004-03-17]. URL http://www.trinux.org

Graham, Robert. (2000). *Sniffing (network wiretap, sniffer) FAQ.* [online document online document 2004-03-01]. URL http://www.robertgraham.com/pubs/sniffing-faq.html

Jacobson, Van., Leres, Craig., & McCanne, Steven. (2003). *The tcpdump man page.* [online document 2004-03-25]. URL http://www.tcpdump.org/tcpdump_man.html

Libvsk. (2000). *Libvsk 1.0.* [online document 2004-04-10]. URL http://www.s0ftpj.org/en/site.html

Linux. (2004). *The Linux Kernel Archives.* [online document 2004-03-30]. URL http://kernel.org/

Mellander, Jim. (2001). *Update – A stealthy Sniffer Detector Part I.* [online document 2004-03-04]. URL http://www.lbl.gov/ITSD/Security/news/mellander_UPDATE-part1.pdf

Vecna. (2000). *BFi numero 9, anno 3.* [online document 2004-04-11]. URL http://www.s0ftpj.org/bfi/online/bfi9/BFi09-14

Welte, Harald. (2000). *The journey of a packet through the linux 2.4 network stack.* [online document 2004-03-25]. URL http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html

Appendices

Appendix 1 – eth.c

Appendix 2 – ip_input.c

Appendix 3 – asp_lkmachk.c

Appendix 4 – fl_aasp.c

Appendix 5 – Questionnaire

Appendix 1 – eth.c

```
/*
 * INET                    An implementation of the TCP/IP protocol suite for the LINUX
 *                         operating system.  INET is implemented using the  BSD Socket
 *                         interface as the means of communication with the user level.
 *
 *                         Ethernet-type device handling.
 *
 * Version:@(#)eth.c       1.0.7         05/25/93
 *
 * Authors:Ross Biro, <bir7@leland.Stanford.Edu>
 *                         Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *                         Mark Evans, <evansmp@uhura.aston.ac.uk>
 *                         Florian  La Roche, <rzsfl@rz.uni-sb.de>
 *                         Alan Cox, <gw4pts@gw4pts.ampr.org>
 *
 * Fixes:
 *                         Mr Linux      : Arp problems
 *                         Alan Cox      : Generic queue tidyup (very tiny here)
 *                         Alan Cox      : eth_header ntohs should be htons
 *                         Alan Cox      : eth_rebuild_header missing an htons and
 *                                                      minor other things.
 *                         Tegge                      : Arp bug fixes.
 *                         Florian                    : Removed many unnecessary functions, code cleanup
 *                                                      and changes for new arp and skbuff.
 *                         Alan Cox      : Redid header building to reflect new format.
 *                         Alan Cox      : ARP only when compiled with CONFIG_INET
 *                         Greg Page     : 802.2 and SNAP stuff.
 *                         Alan Cox      : MAC layer pointers/new format.
 *                         Paul Gortmaker             : eth_copy_and_sum shouldn't csum padding.
 *                         Alan Cox      : Protect against forwarding explosions with
 *                                                      older network drivers and IFF_ALLMULTI.
 *          Christer Weinigel          : Better rebuild header message.
 *              Andrew Morton    : 26Feb01: kill ether_setup() - use netdev_boot_setup().
 *
 *                         This program is free software; you can redistribute it and/or
 *                         modify it under the terms of the GNU General Public License
 *                         as published by the Free Software Foundation; either version
 *                         2 of the License, or (at your option) any later version.
 */
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/socket.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/ip.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/errno.h>
#include <linux/config.h>
#include <linux/init.h>
#include <net/dst.h>
#include <net/arp.h>
#include <net/sock.h>
#include <net/ipv6.h>
#include <net/ip.h>
#include <asm/uaccess.h>
#include <asm/system.h>
#include <asm/checksum.h>

extern int __init netdev_boot_setup(char *str);

__setup("ether=", netdev_boot_setup);

/*
 *           Create the Ethernet MAC header for an arbitrary protocol layer
 *
 *          saddr=NULL    means use device source address
 *          daddr=NULL    means leave destination address (eg unresolved arp)
 */

int eth_header(struct sk_buff *skb, struct net_device *dev, unsigned short type,
              void *daddr, void *saddr, unsigned len)
{
        struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);

        /*
         *              Set the protocol type. For a packet of type ETH_P_802_3 we put the length
         *              in here instead. It is up to the 802.2 layer to carry protocol information.
         */

        if(type!=ETH_P_802_3)
                eth->h_proto = htons(type);
        else
                eth->h_proto = htons(len);

        /*
         *              Set the source hardware address.
         */

        if(saddr)
                memcpy(eth->h_source,saddr,dev->addr_len);
        else
                memcpy(eth->h_source,dev->dev_addr,dev->addr_len);

        /*
         *              Anyway, the loopback-device should never use this function...
         */

        if (dev->flags & (IFF_LOOPBACK|IFF_NOARP))
        {
                memset(eth->h_dest, 0, dev->addr_len);
                return(dev->hard_header_len);
        }

        if(daddr)
        {
                memcpy(eth->h_dest,daddr,dev->addr_len);
                return dev->hard_header_len;
        }

        return -dev->hard_header_len;
}


/*
 *          Rebuild the Ethernet MAC header. This is called after an ARP
 *          (or in future other address resolution) has completed on this
 *          sk_buff. We now let ARP fill in the other fields.
 *
 *          This routine CANNOT use cached dst->neigh!
 *          Really, it is used only when dst->neigh is wrong.
 */

int eth_rebuild_header(struct sk_buff *skb)
{
        struct ethhdr *eth = (struct ethhdr *)skb->data;
        struct net_device *dev = skb->dev;

        switch (eth->h_proto)
        {
#ifdef CONFIG_INET
        case __constant_htons(ETH_P_IP):
                return arp_find(eth->h_dest, skb);
#endif
        default:
                printk(KERN_DEBUG
                       "%s: unable to resolve type %X addresses.\n",
                       dev->name, (int)eth->h_proto);

                memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
                break;
        }

        return 0;
```

```
}


/*
 *      Determine the packet's protocol ID. The rule here is that we
 *      assume 802.3 if the type field is short enough to be a length.
 *      This is normal practice and works for any 'now in use' protocol.
 */

unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)
{
        struct ethhdr *eth;
        unsigned char *rawp;

        skb->mac.raw=skb->data;
        skb_pull(skb,dev->hard_header_len);
        eth= skb->mac.ethernet;

        if(*eth->h_dest&1)
        {
                if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
                        skb->pkt_type=PACKET_BROADCAST;
                else
                        skb->pkt_type=PACKET_MULTICAST;
        }

        /*
         *      This ALLMULTI check should be redundant by 1.4
         *      so don't forget to remove it.
         *
         *      Seems, you forgot to remove it. All silly devices
         *      seems to set IFF_PROMISC.
         */

        else if(1 /*dev->flags&IFF_PROMISC*/)
        {
                if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
                        skb->pkt_type=PACKET_OTHERHOST;
        }

        if (ntohs(eth->h_proto) >= 1536)
                return eth->h_proto;

        rawp = skb->data;

        /*
         *      This is a magic hack to spot IPX packets. Older Novell breaks
         *      the protocol design and runs IPX over 802.3 without an 802.2 LLC
         *      layer. We look for FFFF which isn't a used 802.2 SSAP/DSAP. This
         *      won't work for fault tolerant netware but does for the rest.
         */
        if (*(unsigned short *)rawp == 0xFFFF)
                return htons(ETH_P_802_3);

        /*
         *      Real 802.2 LLC
         */
        return htons(ETH_P_802_2);
}

int eth_header_parse(struct sk_buff *skb, unsigned char *haddr)
{
        struct ethhdr *eth = skb->mac.ethernet;
        memcpy(haddr, eth->h_source, ETH_ALEN);
        return ETH_ALEN;
}

int eth_header_cache(struct neighbour *neigh, struct hh_cache *hh)
{
        unsigned short type = hh->hh_type;
        struct ethhdr *eth = (struct ethhdr*)(((u8*)hh->hh_data) + 2);
        struct net_device *dev = neigh->dev;

        if (type == __constant_htons(ETH_P_802_3))
                return -1;

        eth->h_proto = type;
```

```
           memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
           memcpy(eth->h_dest, neigh->ha, dev->addr_len);
           hh->hh_len = ETH_HLEN;
           return 0;
}

/*
 * Called by Address Resolution module to notify changes in address.
 */

void eth_header_cache_update(struct hh_cache *hh, struct net_device *dev, unsigned char * haddr)
{
           memcpy(((u8*)hh->hh_data) + 2, haddr, dev->addr_len);
}
```

Appendix 2 – ip_input.c

```
/*
 * INET                  An implementation of the TCP/IP protocol suite for the LINUX
 *                       operating system.  INET is implemented using the  BSD Socket
 *                       interface as the means of communication with the user level.
 *
 *                       The Internet Protocol (IP) module.
 *
 * Version:$Id: ip_input.c,v 1.53 2000/12/18 19:01:50 davem Exp $
 *
 * Authors:Ross Biro, <bir7@leland.Stanford.Edu>
 *                       Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *                       Donald Becker, <becker@super.org>
 *                       Alan Cox, <Alan.Cox@linux.org>
 *                       Richard Underwood
 *                       Stefan Becker, <stefanb@yello.ping.de>
 *                       Jorge Cwik, <jorge@laser.satlink.net>
 *                       Arnt Gulbrandsen, <agulbra@nvg.unit.no>
 *
 *
 * Fixes:
 *                       Alan Cox      :            Commented a couple of minor bits of surplus code
 *                       Alan Cox      :            Undefining IP_FORWARD doesn't include the code
 *                                                                (just stops a compiler warning).
 *                       Alan Cox      :            Frames with >=MAX_ROUTE record routes, strict routes or loose
routes
 *                                                                are junked rather than corrupting things.
 *                       Alan Cox      :            Frames to bad broadcast subnets are dumped
 *                                                                We used to process them non broadcast and
 *                                                                boy could that cause havoc.
 *                       Alan Cox      :            ip_forward sets the free flag on the
 *                                                                new frame it queues. Still crap because
 *                                                                it copies the frame but at least it
 *                                                                doesn't eat memory too.
 *                       Alan Cox      :            Generic queue code and memory fixes.
 *                       Fred Van Kempen :          IP fragment support (borrowed from NET2E)
 *                       Gerhard Koerting:          Forward fragmented frames correctly.
 *                       Gerhard Koerting:          Fixes to my fix of the above 8-).
 *                       Gerhard Koerting:          IP interface addressing fix.
 *                       Linus Torvalds             :             More robustness checks
 *                       Alan Cox      :            Even more checks: Still not as robust as it ought to be
 *                       Alan Cox      :            Save IP header pointer for later
 *                       Alan Cox      :            ip option setting
 *                       Alan Cox      :            Use ip_tos/ip_ttl settings
 *                       Alan Cox      :            Fragmentation bogosity removed
 *                                                                (Thanks to Mark.Bush@prg.ox.ac.uk)
 *                       Dmitry Gorodchanin :       Send of a raw packet crash fix.
 *                       Alan Cox      :            Silly ip bug when an overlength
 *                                                                fragment turns up. Now frees the
 *                                                                queue.
 *                       Linus Torvalds/ :          Memory leakage on fragmentation
 *                       Alan Cox      :            handling.
 *                       Gerhard Koerting:          Forwarding uses IP priority hints
 *                       Teemu Rantanen             :             Fragment problems.
 *                       Alan Cox      :            General cleanup, comments and reformat
 *                       Alan Cox      :            SNMP statistics
 *                       Alan Cox      :            BSD address rule semantics. Also see
 *                                                                UDP as there is a nasty checksum issue
 *                                                                if you do things the wrong way.
 *                       Alan Cox      :            Always defrag, moved IP_FORWARD to the config.in file
 *                       Alan Cox      :            IP options adjust sk->priority.
 *                       Pedro Roque   :            Fix mtu/length error in ip_forward.
 *                       Alan Cox      :            Avoid ip_chk_addr when possible.
 *         Richard Underwood           :            IP multicasting.
 *                       Alan Cox      :            Cleaned up multicast handlers.
 *                       Alan Cox      :            RAW sockets demultiplex in the BSD style.
 *                       Gunther Mayer :            Fix the SNMP reporting typo
 *                       Alan Cox      :            Always in group 224.0.0.1
 *         Pauline Middelink           :            Fast ip_checksum update when forwarding
 *                                                                Masquerading support.
 *                       Alan Cox      :            Multicast loopback error for 224.0.0.1
 *                       Alan Cox      :            IP_MULTICAST_LOOP option.
 *                       Alan Cox      :            Use notifiers.
 *                       Bjorn Ekwall  :            Removed ip_csum (from slhc.c too)
 *                       Bjorn Ekwall  :            Moved ip_fast_csum to ip.h (inline!)
 *                       Stefan Becker   :       Send out ICMP HOST REDIRECT
 *         Arnt Gulbrandsen            :            ip_build_xmit
 *                       Alan Cox      :            Per socket routing cache
 *                       Alan Cox      :            Fixed routing cache, added header cache.
 *                       Alan Cox      :            Loopback didn't work right in original ip_build_xmit - fixed it.
 *                       Alan Cox      :            Only send ICMP_REDIRECT if src/dest are the same net.
 *                       Alan Cox      :            Incoming IP option handling.
 *                       Alan Cox      :            Set saddr on raw output frames as per BSD.
 *                       Alan Cox      :            Stopped broadcast source route explosions.
 *                       Alan Cox      :            Can disable source routing
 *                       Takeshi Sone    :          Masquerading didn't work.
 *         Dave Bonn,Alan Cox          :            Faster IP forwarding whenever possible.
 *                       Alan Cox      :            Memory leaks, tramples, misc debugging.
 *                       Alan Cox      :            Fixed multicast (by popular demand 8))
 *                       Alan Cox      :            Fixed forwarding (by even more popular demand 8))
 *                       Alan Cox      :            Fixed SNMP statistics [I think]
 *         Gerhard Koerting            :            IP fragmentation forwarding fix
 *                       Alan Cox      :            Device lock against page fault.
 *                       Alan Cox      :            IP_HDRINCL facility.
 *         Werner Almesberger          :            Zero fragment bug
 *                       Alan Cox      :            RAW IP frame length bug
 *                       Alan Cox      :            Outgoing firewall on build_xmit
 *                       A.N.Kuznetsov :            IP_OPTIONS support throughout the kernel
 *                       Alan Cox      :            Multicast routing hooks
 *                       Jos Vos                    :            Do accounting *before* call_in_firewall
 *         Willy Konynenberg           :            Transparent proxying support
 *
 *
 *
 * To Fix:
 *                       IP fragmentation wants rewriting cleanly. The RFC815 algorithm is much more efficient
 *                       and could be made very efficient with the addition of some virtual memory hacks to permit
 *                       the allocation of a buffer that can then be 'grown' by twiddling page tables.
 *                       Output fragmentation wants updating along with the buffer management to use a single
 *                       interleaved copy algorithm so that fragmenting has a one copy overhead. Actual packet
 *                       output should probably do its own fragmentation at the UDP/RAW layer. TCP shouldn't cause
 *                       fragmentation anyway.
 *
 *                       This program is free software; you can redistribute it and/or
 *                       modify it under the terms of the GNU General Public License
 *                       as published by the Free Software Foundation; either version
 *                       2 of the License, or (at your option) any later version.
 */

#include <asm/system.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/config.h>

#include <linux/net.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>

#include <net/snmp.h>
#include <net/ip.h>
#include <net/protocol.h>
#include <net/route.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/arp.h>
#include <net/icmp.h>
#include <net/raw.h>
#include <net/checksum.h>
#include <linux/netfilter_ipv4.h>
#include <linux/mroute.h>
#include <linux/netlink.h>

/*
 *         SNMP management statistics
 */

struct ip_mib ip_statistics[NR_CPUS*2];

/*
 *      Process Router Attention IP option
 */
int ip_call_ra_chain(struct sk_buff *skb)
{
        struct ip_ra_chain *ra;
        u8 protocol = skb->nh.iph->protocol;
        struct sock *last = NULL;

        read_lock(&ip_ra_lock);
        for (ra = ip_ra_chain; ra; ra = ra->next) {
                struct sock *sk = ra->sk;

                /* If socket is bound to an interface, only report
                 * the packet if it came  from that interface.
                 */
                if (sk && sk->num == protocol
                    && ((sk->bound_dev_if == 0)
                        || (sk->bound_dev_if == skb->dev->ifindex))) {
                        if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
                                skb = ip_defrag(skb);
                                if (skb == NULL) {
                                        read_unlock(&ip_ra_lock);
                                        return 1;
                                }
                        }
                        if (last) {
                                struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
                                if (skb2)
                                        raw_rcv(last, skb2);
                        }
                        last = sk;
                }
        }

        if (last) {
                raw_rcv(last, skb);
                read_unlock(&ip_ra_lock);
                return 1;
        }
        read_unlock(&ip_ra_lock);
        return 0;
}

/* Handle this out of line, it is rare. */
static int ip_run_ipprot(struct sk_buff *skb, struct iphdr *iph,
                         struct inet_protocol *ipprot, int force_copy)
{
        int ret = 0;

        do {
                if (ipprot->protocol == iph->protocol) {
                        struct sk_buff *skb2 = skb;
                        if (ipprot->copy || force_copy)
                                skb2 = skb_clone(skb, GFP_ATOMIC);
                        if(skb2 != NULL) {
                                ret = 1;
                                ipprot->handler(skb2);
                        }
                }
                ipprot = (struct inet_protocol *) ipprot->next;
        } while(ipprot != NULL);

        return ret;
}

static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
        int ihl = skb->nh.iph->ihl*4;

#ifdef CONFIG_NETFILTER_DEBUG
        nf_debug_ip_local_deliver(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/
```

```
           /* Pull out additionl 8 bytes to save some space in protocols. */
           if (!pskb_may_pull(skb, ihl+8))
                         goto out;
           __skb_pull(skb, ihl);

#ifdef CONFIG_NETFILTER
           /* Free reference early: we don't need it any more, and it may
           hold ip_conntrack module loaded indefinitely. */
           nf_conntrack_put(skb->nfct);
           skb->nfct = NULL;
#endif /*CONFIG_NETFILTER*/

        /* Point into the IP datagram, just past the header. */
        skb->h.raw = skb->data;

           {
                         /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
                         int protocol = skb->nh.iph->protocol;
                         int hash = protocol & (MAX_INET_PROTOS - 1);
                         struct sock *raw_sk = raw_v4_htable[hash];
                         struct inet_protocol *ipprot;
                         int flag;

                         /* If there maybe a raw socket we must check - if not we
                          * don't care less
                          */
                         if(raw_sk != NULL)
                                      raw_sk = raw_v4_input(skb, skb->nh.iph, hash);

                         ipprot = (struct inet_protocol *) inet_protos[hash];
                         flag = 0;
                         if(ipprot != NULL) {
                                      if(raw_sk == NULL &&
                                         ipprot->next == NULL &&
                                         ipprot->protocol == protocol) {
                                                    int ret;

                                                    /* Fast path... */
                                                    ret = ipprot->handler(skb);

                                                    return ret;
                                      } else {
                                                    flag = ip_run_ipprot(skb, skb->nh.iph, ipprot, (raw_sk != NULL));
                                      }
                         }

                         /* All protocols checked.
                          * If this packet was a broadcast, we may *not* reply to it, since that
                          * causes (proven, grin) ARP storms and a leakage of memory (i.e. all
                          * ICMP reply messages get queued up for transmission...)
                          */
                         if(raw_sk != NULL) {       /* Shift to last raw user */
                                      raw_rcv(raw_sk, skb);
                                      sock_put(raw_sk);
                         } else if (!flag) {                      /* Free and report errors */
                                      icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
out:
                                      kfree_skb(skb);
                         }
           }

           return 0;
}

/*
 *         Deliver IP Packets to the higher protocol layers.
 */
int ip_local_deliver(struct sk_buff *skb)
{
           /*
            *            Reassemble IP fragments.
            */

           if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
                         skb = ip_defrag(skb);
                         if (!skb)
```

```
                        return 0;
        }

        return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                    ip_local_deliver_finish);
}

static inline int ip_rcv_finish(struct sk_buff *skb)
{
        struct net_device *dev = skb->dev;
        struct iphdr *iph = skb->nh.iph;

        /*
         *      Initialise the virtual path cache for the packet. It describes
         *      how the packet travels inside Linux networking.
         */
        if (skb->dst == NULL) {
                if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
                        goto drop;
        }

#ifdef CONFIG_NET_CLS_ROUTE
        if (skb->dst->tclassid) {
                struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
                u32 idx = skb->dst->tclassid;
                st[idx&0xFF].o_packets++;
                st[idx&0xFF].o_bytes+=skb->len;
                st[(idx>>16)&0xFF].i_packets++;
                st[(idx>>16)&0xFF].i_bytes+=skb->len;
        }
#endif

        if (iph->ihl > 5) {
                struct ip_options *opt;

                /* It looks as overkill, because not all
                   IP options require packet mangling.
                   But it is the easiest for now, especially taking
                   into account that combination of IP options
                   and running sniffer is extremely rare condition.
                                                      --ANK (980813)
                */

                if (skb_cow(skb, skb_headroom(skb)))
                        goto drop;
                iph = skb->nh.iph;

                skb->ip_summed = 0;
                if (ip_options_compile(NULL, skb))
                        goto inhdr_error;

                opt = &(IPCB(skb)->opt);
                if (opt->srr) {
                        struct in_device *in_dev = in_dev_get(dev);
                        if (in_dev) {
                                if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                                        if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
                                                printk(KERN_INFO "source route option
%u.%u.%u.%u -> %u.%u.%u.%u\n",
                                                       NIPQUAD(iph->saddr),
NIPQUAD(iph->daddr));
                                        in_dev_put(in_dev);
                                        goto drop;
                                }
                                in_dev_put(in_dev);
                        }
                        if (ip_options_rcv_srr(skb))
                                goto drop;
                }
        }

        return skb->dst->input(skb);

inhdr_error:
        IP_INC_STATS_BH(IpInHdrErrors);
drop:
        kfree_skb(skb);
        return NET_RX_DROP;
}

/*
 *         Main IP Receive routine.
 */
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
{
           struct iphdr *iph = skb->nh.iph;

           /* When the interface is in promisc. mode, drop all the crap
            * that it receives, do not try to analyse it.
            */
           if (skb->pkt_type == PACKET_OTHERHOST)
                         goto drop;

           IP_INC_STATS_BH(IpInReceives);

           if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
                         goto out;

           if (!pskb_may_pull(skb, sizeof(struct iphdr)))
                         goto inhdr_error;

           iph = skb->nh.iph;

           /*
            *            RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails the checksum.
            *
            *            Is the datagram acceptable?
            *
            *            1.            Length at least the size of an ip header
            *            2.            Version of 4
            *            3.            Checksums correctly. [Speed optimisation for later, skip loopback checksums]
            *            4.            Doesn't have a bogus length
            */

           if (iph->ihl < 5 || iph->version != 4)
                         goto inhdr_error;

           if (!pskb_may_pull(skb, iph->ihl*4))
                         goto inhdr_error;

           if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
                         goto inhdr_error;

           {
                         __u32 len = ntohs(iph->tot_len);
                         if (skb->len < len || len < (iph->ihl<<2))
                                       goto inhdr_error;

                         /* Our transport medium may have padded the buffer out. Now we know it
                          * is IP we can trim to the true length of the frame.
                          * Note this now means skb->len holds ntohs(iph->tot_len).
                          */
                         if (skb->len > len) {
                                       __pskb_trim(skb, len);
                                       if (skb->ip_summed == CHECKSUM_HW)
                                                     skb->ip_summed = CHECKSUM_NONE;
                         }
           }

           return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
                                ip_rcv_finish);

inhdr_error:
           IP_INC_STATS_BH(IpInHdrErrors);
drop:
        kfree_skb(skb);
out:
        return NET_RX_DROP;
}
```

Appendix 3 – asp_lkmachk.c

```c
/*
# gcc -O6 -c aasp_lkmachk.c -I/usr/src/linux/include
# insmod aasp_lkmachk.o device=eth0
# rmmod aasp_lkmachk

Anti Anti Sniffer Patch (by vecna@s0ftpj.org) - MAC checker module

*/

#define MODULE
#define __KERNEL__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>

#include <linux/byteorder/generic.h>
/* on kernel 2.2.16 I've find some problem and for fix I've cut inclusion
   of generic.h
*/
#include <linux/netdevice.h>
#include <net/protocol.h>
#include <net/pkt_sched.h>
#include <net/tcp.h>
#include <net/ip.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/skbuff.h>

#include <linux/kernel.h>
#include <linux/mm.h>
#include <linux/file.h>
#include <asm/uaccess.h>

#define r_mac sk->mac.ethernet->h_dest  /* received mac */
#define t_mac true->dev_addr                        /* true mac */

char *device;
MODULE_PARM(device, "s");

struct device *true;
struct packet_type aasp_ip, aasp_arp;

int chk_mac_arp(struct sk_buff *sk, struct device *dev, struct packet_type *pt)              {

           if( r_mac[0] ==r_mac[1] ==r_mac[2] ==r_mac[3] ==r_mac[4]
               ==r_mac[5] ==0xff)
                         /* mac broadcast */
                         goto end;

           if( (r_mac[0] !=t_mac[0]) || (r_mac[1] !=t_mac[1]) ||
               (r_mac[2] !=t_mac[2]) || (r_mac[3] !=t_mac[3]) ||
               (r_mac[4] !=t_mac[4]) || (r_mac[5] !=t_mac[5]) )
                         {
                         /* ARP mac spoof detected */
                         sk->nh.arph->ar_hrd = 0;
                         sk->nh.arph->ar_pro = 0;
                         sk->nh.arph->ar_op = 0;
                         goto end;
                         }
end:
           kfree_skb(sk);
           return(0);
           }

int chk_mac_ip(struct sk_buff *sk, struct device *dev, struct packet_type *pt)
           {

           if( r_mac[0] ==r_mac[1] ==r_mac[2] ==r_mac[3] ==r_mac[4]
               ==r_mac[5] ==0xff)
                         /* mac broadcast*/
                         goto end;

           if( (r_mac[0] !=t_mac[0]) || (r_mac[1] !=t_mac[1]) ||
               (r_mac[2] !=t_mac[2]) || (r_mac[3] !=t_mac[3]) ||
               (r_mac[4] !=t_mac[4]) || (r_mac[5] !=t_mac[5]) )
                         {
                         /* IP check - anti spoof detect! */
                         sk->nh.iph->tot_len = 0;
                         sk->nh.iph->check = 0;
                         goto end;
                         }
end:
           kfree_skb(sk);
           return(0);
           }

int init_module(void)
           {

           if (device)
                         {
                         true =dev_get(device);
                         if (true ==NULL)
                                       {
                                       printk("Did not find device %s!\n", device);
                                       return -EINVAL;
                                       }
                         }
           else
                         {
                         printk("Usage: insmod aasp_lkmachk.o device=device name \n\n");
                         return -ENODEV;
                         }

           printk("Mac checker module run on %s - by vecna@s0ftpj.org\n",device);
           printk("Full codes of Anti Anti Sniffer Patch can be"
                         " downloadated at www.s0ftpj.org\n");

           aasp_ip.dev = true;
           aasp_ip.type = htons(ETH_P_IP);
           aasp_ip.func = chk_mac_ip;

           aasp_arp.dev = true;
           aasp_arp.type = htons(ETH_P_ARP);
           aasp_arp.func = chk_mac_arp;

           dev_add_pack(&aasp_ip);
           dev_add_pack(&aasp_arp);

           return(0);
           }

void cleanup_module(void)
           {
           dev_remove_pack(&aasp_ip);
           dev_remove_pack(&aasp_arp);
           printk("Anti Anti Sniffer Patch - MAC checker module unload\n");
           }
```

Appendix 4 – fl_aasp.c

```c
/*
Fucker Latency test for Anti Anti Sniffer Patch
*/

#include "libvsk.h" /* www.s0ftpj.org for more info */
#include <errno.h>

extern int errno;

#define fatal(M)         {               \
                                      perror(M); \
                                      exit(0);   \
                                      }

#define IPSIZE                        sizeof(struct iphdr)
#define ICMPSIZE         sizeof(struct icmphdr)
#define IIPKTSIZE        sizeof(struct iipkt)

int check_dup(struct iipkt *);
void build_reply(struct iipkt *, struct sockaddr_in *, struct iipkt *);
unsigned short ip_s(unsigned short *, int);

int main(int argc, char **argv)
           {
           int dlsfd, offset, forward, hdrincl =1, pkt_info[4], x;
           char ipdst[18], *rcvd =malloc(IIPKTSIZE);
           struct ifreq ifr;
           struct in_addr in;
           struct iipkt *reply =malloc(IIPKTSIZE);

           printf("\t Anti Anti Sniffer Patch for elude latency test\n");
           printf("\t by vecna - vecna@s0ftpj.org - www.s0ftpj.org\n\n");

           if(argc != 3)
                         {
                         printf( " usage %s interface fakedelay\n\n", argv[0]);
                         exit(0);
                         }

           printf(" running on background\n");
           if(fork())
                         exit(0);

           pkt_info[0] =pkt_info[1] =ICMP_ECHO;
           pkt_info[2] =0;
           pkt_info[3] =0xFFFF;

           x =socket(PF_INET, SOCK_DGRAM, IPPROTO_IP);

           strncpy(ifr.ifr_name, argv[1], sizeof(ifr.ifr_name));
           if(ioctl (x, SIOCGIFADDR, &ifr) < 0)
                         fatal("unable to look local address");

           memcpy((void *)&in, (void *)&ifr.ifr_addr.sa_data +2, 4);
           strcpy(ipdst, (char *)inet_ntoa(in));
           close(x);

           dlsfd =set_vsk_param(NULL, ipdst, pkt_info, argv[1],
                                                   IPPROTO_ICMP, IO_IN, IP_FW_INSERT, 0, 0);
           if(dlsfd < 0)
                         fatal("set_vsk: IP_FW_INSERT");

           if((offset =get_offset(dlsfd, argv[1])) <0)
                         fatal("get device offset");

           if((forward = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1)
                         fatal("forward socket - SOCK_RAW");

           if((x = setsockopt(forward, IPPROTO_IP, IP_HDRINCL,
                                                                &hdrincl, sizeof(hdrincl))) == -1)
                         fatal("setsockopt - IP_HDRINCL");
           while(1)
                         {
                         struct iipkt *packet;
                static int last_id;

                         read(dlsfd, rcvd, IIPKTSIZE);

                         (char *)packet = rcvd + offset;

                         if(check_dup(packet))
                                       continue;

                         if(check_packet(packet, IPPROTO_ICMP))
                                       {
                                       struct sockaddr_in sin;

                                       build_reply(packet, &sin, reply);

                                       usleep(atoi(argv[2]));
                                       /*
                                       poll & select it's more intelligent...
                                       mah... maybe
                                       */

                                       x =sendto(forward, (char *)reply,
                                                    ntohs(reply->ip.tot_len), 0,
                                                    (struct sockaddr *)&sin,
                                                    sizeof(struct sockaddr)    );

                                       if(x < 0)
                                                    fatal("sendto on forwarding packet");

                                       }
                         memset(packet, 0, IIPKTSIZE);
                         }
           free(rcvd);   /* never here */
           }

void build_reply(struct iipkt *packet, struct sockaddr_in *sin,
                                                                                struct iipkt *reply)
           {

           memcpy((void *)reply, (void *)packet, IIPKTSIZE);

           reply->ip.id =getpid() & 0xffff ^ packet->ip.id;
           reply->ip.saddr =packet->ip.daddr;
           reply->ip.daddr =packet->ip.saddr;
           reply->ip.check            =ip_s((u_short *)&reply->ip, IPSIZE);

           reply->icmp.type =ICMP_ECHOREPLY;
           reply->icmp.checksum =0x0000;
           reply->icmp.checksum =ip_s((u_short *)&reply->icmp,
                                                    ntohs(packet->ip.tot_len) - IPSIZE );

           /* setting sockaddr_in stuctures */
           sin->sin_port =htons(0);
           sin->sin_family = AF_INET;
           sin->sin_addr.s_addr = reply->ip.daddr;
           }

int check_dup(struct iipkt *packet)
           {
           static int last_id;
           int id =htons(packet->ip.id);

           if(id ==htons(last_id))
                         return 1;

           last_id =packet->ip.id;

           return 0;
           }

u_short ip_s(u_short *ptr, int nbytes)
           {
           register long sum = 0;
```

```
u_short oddbyte;
register u_short answer;

while (nbytes > 1)
              {
              sum += *ptr++;
              nbytes -= 2;
              }
if (nbytes == 1)
              {
              oddbyte = 0;
              *((u_char *) &oddbyte) = *(u_char *)ptr;
              sum += oddbyte;
              }
sum  = (sum >> 16) + (sum & 0xffff);
sum += (sum >> 16);
answer = ~sum;

return(answer);
}
```

Appendix 5 – Questionnaire

Do you find Network-based non-standalone sniffer detection reliable?

Answer (Yes/No):

Do you think counter detection methods pose a threat to the reliability of Network-based non-standalone sniffer detection?

Answer (Yes/No):

Do you think that the very nature of Network-based non-standalone sniffer detection makes it unfeasible to develop reliable detection methods?

Answer (Yes/No):

Do you think that the Open Source nature of Linux, affects the feasibility of implementation of counter detection methods against Network-based non-standalone sniffer detection?

Answer (Yes/No):

Which type of Network-based non-standalone sniffer detection method do you find most reliable?

1.) MAC-based methods
2.) Decoy-based methods
3.) Network and machine latency based methods
4.) None of the above

Answer (1-4):