Remote Sniffer Detection

David Wu and Frederick Wong
{davidwu, fredwong}@cs.berkeley.edu

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

December 14, 1998

Abstract

Sniffers are common tools of the hacker trade. Typically, hackers gain entry to a host machine on a network and install a password sniffer to obtain passwords which they hope will allow them to compromise other hosts. Since sniffers are so widely used, the ability to detect when sniffers are running on a network can be a tremendous help in fighting off hacker attacks. Unfortunately, there is currently very little help for the system administrator to detect the presence of sniffers on the network. Other than logging in to all the machines on the network and checking if a sniffer is running (and in most cases, hackers can hide their trails), the system administrator is left with no other tools to help them find sniffers. This paper presents the design and implementation of tools which can give system administrators a helping hand. These tools detect when a sniffer is running on a Ethernet network remotely, without logging in to the machines.

1 INTRODUCTION

Sniffers are programs which allow eavesdropping on share medium networks such as Ethernet [DIX80]. Sniffers are run on a machine that is connected to the network and capture the network traffic. Normally, sniffers are used for debugging network problems. However, sniffers have an alternative use that hackers prefer. Sniffers can be used to capture passwords sent in clear text. Clear text passwords are relatively common in the TCP [Pos81-2]/IP [Pos81-1] protocol suite. Basic services such as telnet [PR83], rlogin [Kan91], ftp [PR85], and pop mail [MR94] authenticates using clear text passwords.

The use of sniffers for capturing passwords are so useful to hackers that specialized sniffers that only capture passwords have been created. Password sniffers such as `esniff` and `linsniff` are readily available at the local hacker shops for use on broken-in Unix machines.[1] Furthermore, more traditional sniffers such as `tcpdump` [JLM89] and `sniffit` can be used to sniff passwords as well.

Aside from the obvious negative effect of having passwords stolen and machines compromised, password sniffing can cause a huge drain on machine resources. Network sniffing can take up large amounts of processing power on a heavily loaded network. Sniffer logs can build quickly and eat up large amounts of disk space especially if the sniffer is poorly written and has bugs. Thus, loss of processing power and hard drives suddenly becoming full are usually good indications that a hacker has broken in. However, hard drives take time to full up (if at all), and machine load varies from legitimate uses. Hackers also are usually clever enough to hide the sniffers from the `ps` command. Furthermore, the busy system administrator usually does not even have the time to randomly log in to machines and run a `ps` and check if the load corroborates with the legitimate uses of the machine.

In this paper, we will present the design and implementation of three different techniques which can detect when a sniffer is running on an Ethernet network. These techniques were im-

[1] `http://www.rootshell.com` is a wonderful hacker shop for picking up common hacking tools.

plemented in three tools which can be used to automatically detect when a sniffer is present on a network. This paper is organized as follows. In section 2, we will discuss the motivation of developing these techniques. In section 3, we list the assumptions we have made and explain why they are reasonable. In section 4, we will present the first technique, “MAC detection”. In section 5, we will present the second technique, “DNS detection”. In section 6, we will present the third technique, “Load detection”. We will discuss the differences between the various techniques in section 7. Some plans for future work will be listed in section 8. Lastly, we will conclude in section 9.

2 MOTIVATION

There is an obvious motivation behind why sniffer detection is useful. Password sniffers are one of the most common programs installed by the hackers after intrusion. Password sniffers allow hackers to obtain more passwords in order to compromise more machines. Commonly, a hacker's presence is detected when a password sniffer is found to be running on a compromised machine.

There are a few ways in which a system administrator can detect that a sniffer is running on his network. A system administrator can log in to his machines and check for unusual behavior. These behavior include high machine load and shrinking hard drive space because sniffers are heavy consumers of machine resources. The system administrator can also check what processes are running on the system and the various system and kernel logs. The hacker may not be savvy enough to hide his trails, so kernels that log when a network device is put in to promiscuous mode, which sniffers rely on to perform its duties, can record when a hacker broke in.

However, these techniques are all manual and hard to automate. Furthermore, smarter hackers (or dumb hackers that use better made prepacked hacker tool kits) can usually hide their trails. What is needed is tools which can detect when sniffers are running on a network remotely.

Currently, such tools are virtually nonexistant. [Gru98] is one such tool that attempts to detect when sniffers are running by introducing “bait traffic”, that is fake password traffic which might entice a hacker to log in to a machine that has been set up as a trap. However, the problem with this technique is that the hacker may never take the bait since typically, large amounts of passwords are captured, and the hacker may not chose the trap machine as a target to compromise.

What is needed is tools which can be automated and can detect sniffers remotely. It is necessary to develop techniques that can detect sniffers based on the sniffers programs themselves rather than what a hacker might do.

3 ASSUMPTIONS

We have made various assumptions when we developed our remote sniffer detector. These assumptions limit the types of sniffers that we can detect. However, we feel that our assumptions are valid and reasonable.

One assumption we have made is that the sniffer is an actual sniffer program running on a host. That is, we disallow the possibility that the sniffer is a dedicated device that a hacker physically attaches to the network. This is a rather reasonable assumption since a lot of break-ins are done remotely by hackers with no physical access to the network whatsoever. Usually, a UNIX machine is broken in to, and the hacker logs on to the compromised machine and installs a sniffer with root access.

Another assumption we have made is that the network segment that we are interested in, the network segment which we wish to detect whether a sniffer is running or not, is an Ethernet segment. Again, this is a reasonable assumption since a large percentage of the network segments on the Internet are Ethernet.

This leads us to mention that we also assume that TCP/IP is the protocol that the network is using. Although some of our techniques can be modified to support other networking protocols, the implementation is based on TCP/IP since it is, by far, the most popular network protocol today.

Lastly, the various techniques have their own assumptions. We will state those assumptions in the section the techniques are discussed.

4 MAC DETECTION

The MAC detection technique for detecting sniffers running on a Ethernet segment requires that the machine running the detector be on the same Ethernet segment as the host that is suspected of running a sniffer. Thus, this technique allows remote detection of sniffers on the same Ethernet segment, but not the remote detection of sniffers across different networks.

The basic idea behind the MAC detection technique is simple and has been discussed in the past [Uni97]. However, in order to understand how this technique works, we must make a brief digression on how Ethernet network interface cards work and how TCP/IP is layered on top of Ethernet packets.

4.1 Ethernet Network Interface Cards

A basic Ethernet network interface card has a unique medium access control (MAC) address assigned to it by its manufacturer. Thus, all network interface cards (NIC) can be uniquely identified by its MAC address. Since Ethernet is a shared medium network, all data packets are essentially broadcasted. Since passing all packets broadcasted on the network to the operating system is inefficient, Ethernet controller chips typically implement a filter which filters out any packet that does not contain a target MAC address for the NIC.[2]

Since sniffers are interested in all traffic on the Ethernet segment, NICs provide a promiscuous mode. In promiscuous mode, all Ethernet data packets, regardless of the target MAC address, are passed to the operating system. Thus, when a sniffer is running on a machine, the machine's NIC is set to promiscuous mode to capture all of the Ethernet traffic. Figure 1 shows the flow diagram of the Ethernet data packet path to the operating system.

[2]There are also broadcast data packets, which are always passed to the operating system, as well as multicast data packets, which, depending on the controller chip implementation, can either be always passed to the operating system or passed only if the MAC address passes a hash filter.

4.2 TCP/IP on Ethernet

The Ethernet protocol standard, IEEE 802.3, specifies the Ethernet packet structure. Figure 2 shows a IP packet encapsulated in a Ethernet packet. For TCP/IP, a normal IP packet destined to a particular Ethernet host has the destination host's MAC address filled in the Ethernet header and the IP address of the destination filled in the IP header. Thus, IP packets transported by Ethernet have two addresses, both of which normally correspond to a machine's MAC address and IP address [Hor84] [Plu82].

4.3 Exploiting an Implementation Hole

The MAC detection technique works by exploiting an implementation hole in the TCP/IP stack of some operating systems. On some TCP/IP stacks, the target MAC address of the Ethernet header is never checked. The only check that is made is against the target IP address. This is normally acceptable because no packets should be addressed to a host IP address with the MAC address of some other host. Thus, when a NIC is put in to promiscuous mode, it is possible to generate a IP packet, encapsulated in a Ethernet packet, that contains the correct IP address for a host and an incorrect MAC address (that is, a MAC address that is inconsistent with the host's NIC MAC address) that is passed to the TCP/IP processing code. Therefore, whereas normally, such a packet would never even reach the operating system for processing since the NIC hardware will reject it, on some implementations of TCP/IP, when the NIC is in promiscuous mode, these packets can actually get processed by the TCP/IP stack as if the packet had a correct MAC address.

The trick for this technique, then, is to elicit a response from the TCP/IP stack so that we know the incorrectly addressed packet is acknowledged. Fortunately, the ICMP protocol [Pos94] provides an ICMP Echo Request packet, more commonly known as a "ping request". The semantics of the ICMP Echo Request is that when a host receives the ICMP Echo Request, an ICMP Echo Reply packet is generated. The ICMP Echo Reply packet is destined to the machine that generated the ICMP Echo Request

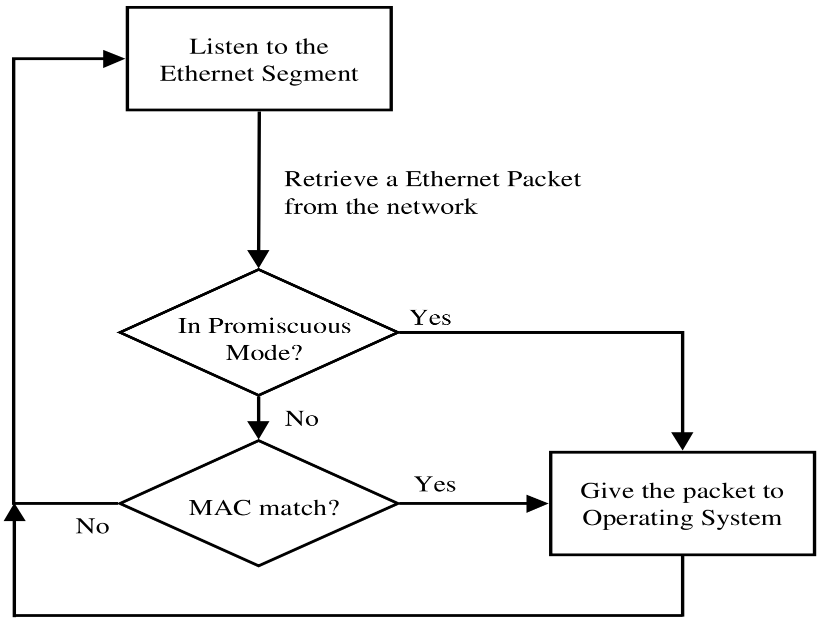

Figure 1: **Flow diagram of the Ethernet data packet path to the operating system** *This diagram shows the control logic performed by the Ethernet controller on the NIC. If the Ethernet NIC is in promiscuous mode, all packets are passed to the operating system regardless of the target MAC address. Otherwise, only correctly MAC addressed packets are passed to the operating system.*

packet.

4.4 Implementation

The implementation of the MAC detection technique is quite simple. The detection tool implements a ICMP Echo Request packet generator. The tool generates the full ICMP packet as well as the outer Ethernet packet that encapsulates the ICMP packet. The Ethernet packet is generate such that the target MAC address is different from the actual MAC address of the target machine. So, for any suspected host on the Ethernet segment, the tool can generate the ICMP Echo Request with incorrect MAC address and check if a ICMP Echo Reply is returned. If so, the suspected host is in promiscuous mode. Thus, a sniffer could likely be running on that host. Figure 3 shows how the MAC detection technique works as implemented.

4.5 Results

The MAC detection technique works only against operating systems with a TCP/IP protocol stack that does not have the check against correct MAC addresses. We were able to confirm that Linux 2.0.35 was vulnerable to this kind of sniffer detection.[3] We were able to detect when a Linux machine went in to promiscuous mode with 100% accuracy. However, FreeBSD 2.2.7 was not vulnerable to this kind of sniffer detection. The networking code in FreeBSD 2.2.7 correctly implements the necessary check so that incorrectly addressed Ethernet packets never reach the ICMP processing code.

5 DNS DETECTION

The DNS detection technique exploits a behavior common in all password sniffers to date. This technique requires that the system ad-

[3] As a side note, we were able to discover a minor bug in the ARP cache code for Linux. Under most Unix systems, a user with root privileges can modify the ARP cache. The ARP cache can be manually set so that a particular IP address maps to a settable MAC address. When this is done, the ARP entry should be locked. That is, all ARP announcements for this IP address should be ignored and the MAC address manually set by the user should be used instead. It turns out that Linux does not lock down the MAC address. If a real host sends out an ARP broadcast, Linux will delete the manually set MAC address and replace it with the announced MAC address.

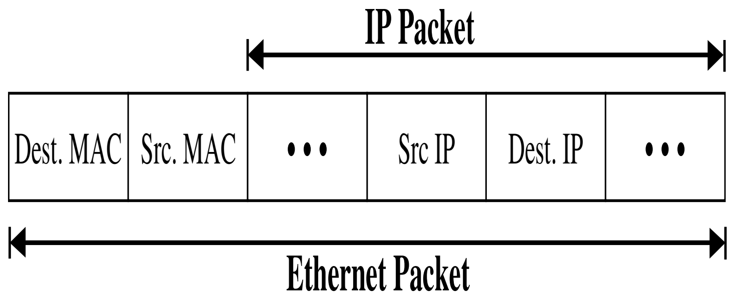

Figure 2: **IP packets encapsulated in an Ethernet packet** *This figure shows an IP packet encapsulated in an Ethernet packet. Note that there are fields for the MAC source and destination addresses in the Ethernet header, while there are fields for IP source and destination addresses in the IP header.*

ministrator controls the Domain Name Server (DNS) [Moc87-1] [Moc87-2] for an unused IP address. Specifically, the system administrator must be registered to provide reverse DNS lookup, that is looking up a hostname by an IP address.

5.1 Exploit Sniffer Behavior

The DNS detection technique works by exploiting a behavior common to all password sniffers we have seen. The key observation is that all current password sniffers are not truly passive. In fact, password sniffers do generate network traffic, although it is usually hard to distinguish whether the generated network traffic was from the sniffer or not. It turns out that all password sniffers we have come across do a reverse DNS lookup on the traffic that it sniffed. Since this traffic is generated by the sniffer program, the trick is to detect this DNS lookup some how and distinguish it from normal DNS lookup requests.

It is not hard to come up with the following idea. We can generate fake traffic to the Ethernet segment with a source address of some unused IP address that we provide the DNS service for. Then, since the traffic we generate should normally be ignored by the hosts on the segment, if a DNS lookup request is generated, we know that there is a sniffer on the Ethernet segment. Figure 4 shows how the DNS technique works.

5.2 Implementation

The implementation of the DNS detection technique is quite straight forward. The tool that implements this technique runs on a machine that is registered to provide the reverse DNS lookup for the trigger IP address, the invalid IP address that is used as the source address in the fake traffic. The tool generates a fake FTP [PR85] connection with the source IP address set to the trigger IP address. Then, the tool waits for a period of user defineable time on the DNS service port. Within this period of time, the tool counts the number of DNS requests for the trigger IP address. When the time expires, the tool reports the number of DNS request counted. Note that the tool never returns a DNS reply. This is to avoid having the DNS entry being cached in some intermediate DNS server.

The reason why DNS request needs to be counted is that the fake FTP traffic may actually be destined for a real machine on the network that provides FTP service. If so, that machine may trigger a DNS lookup. Thus, there are two cases we need to consider. If the fake FTP traffic ends up being destined to a real machine on the network, then if we count two or more DNS lookups, a sniffer is probably running on the network. Otherwise, if only one DNS lookup occurs, it is probably a legitimate lookup being performed by the host. The other case is that the fake traffic ends up being destined to no particular machine on the network. Then, if one or more DNS lookup occurs, there is most likely a sniffer on the network.[4]

[4]Note that typically, for remote sniffer detection across a network, it is usually impossible to generate fake traffic to a non-existant host on the Ethernet segment. This is because an unused IP address in a Ethernet subnet has no MAC address because no host will reply to the ARP

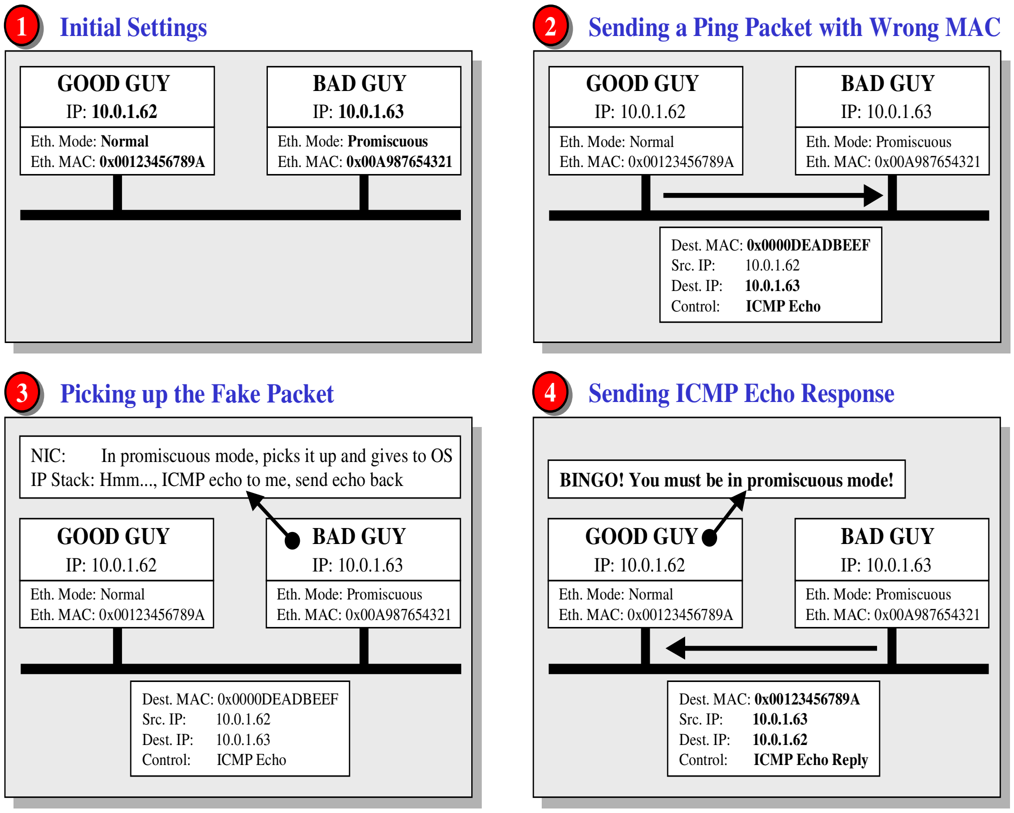

Figure 3: **Diagram for the MAC detection technique** *This figure shows the how the MAC detection technique works.*

5.3 Results

The DNS detection technique was able to detect sniffers running on a Ethernet segment with 100% accuracy regardless of operating system type. The default behavior of `esniff`, `linsniff`, `sniffit`, and even `tcpdump` is to perform the reverse DNS lookup. Furthermore, it is possible to assign a trigger IP address to each network segment to perform the DNS detection technique. This is useful because even if the password sniffer does not perform a reverse DNS lookup, that is, the tool does not detect a sniffer in the required timeout period, the hacker may sometime in the future perform a reverse DNS lookup on the logged password entry. If so, then this technique can be extended to keep track of which IP address is assigned to what network and report a DNS lookup whenever it sees it in the future. Then, the system administrator will be able to tell which network has been compromised.

The timeout period for this technique should be considered carefully. The problem is that we need to avoid DNS lookups that timeout at the host and cause another DNS lookup request to be generated, while allowing for enough time for DNS lookup requests to reach the tool. It currently defaults to 255 seconds since the maximum time to live (TTL) on Internet packets is 255. We have not determined the best timeout period, although 255 seconds seems to work reasonably well. In retrospect, the tool should just return a DNS reply with a TTL set to 0 on the DNS entry, which would prevent a retransmit of the DNS lookup request.

The drawback of this technique is that it is generally not possible to detect what machine is running the sniffer. The tool can be extended to detect which host is running the sniffer if it can sniff the packets on the Ethernet segment to check who is sending the DNS request, however,

request. Thus, the router will never generate the traffic on the network. However, this is possible to do if the machine running the tool is on the same network, therefore it can generate the fake traffic with invalid MAC addresses.

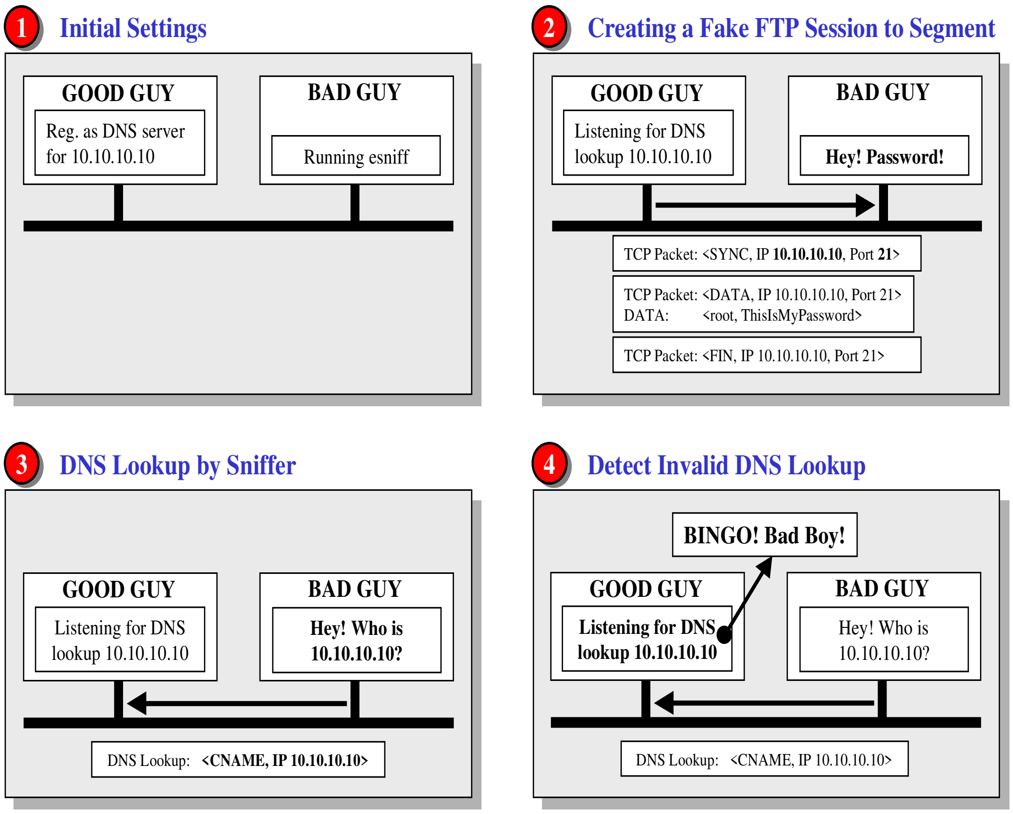

Figure 4: **Diagram for the DNS detection technique** *This figure shows how the DNS detection technique works.*

that means that sniffer detection cannot be done across a router. Or, the tool can be extended to detect which host is running the sniffer if all the hosts on the network set its DNS server to the machine running the tool. The tool can then tell who made the DNS request to the trigger IP address (while passing on other DNS requests to a real DNS server). However, as currently implemented, this technique can at least detect how many sniffers are possibly running on the network segment. For example, if a real host is reached by the fake traffic, the number of request minus one is the number of sniffers running on the network segment.

This technique is in some sense related to the technique used in [Gru98]. In [Gru98], a fake telnet session is generated to a host that is not publicized. This unpublicized host runs a telnet service that logs the incoming connections. Thus, if a telnet connection is ever made to this host, it is probable that hacker has used a sniffer on the network. This technique is related to ours in the sense that "bait traffic" is used to lure sniffers in to revealing themselves. In [Gru98], the bait traffic leads to a telnet server, while in our technique, the bait traffic, the trigger IP address, leads to a DNS server.

Our technique has several advantages, however. First of all, our technique detects a sniffer when it is running, whereas in [Gru98], the sniffer is detected only when the bait is taken, if it is ever taken. By the time the bait is taken, the sniffer may have compromised quite a bit more machines. Furthermore, our technique depends on the sniffer program behavior, which, fortunately, all current password sniffers exhibit this program behavior. In [Gru98], the technique depends on whether the hacker deems the host interesting enough to warrant a visit. However, the disadvantage is that our technique can be quickly side stepped. Sniffers can easily be changed to not perform the reverse DNS lookup. Furthermore, hackers will become more intelligent so as to never perform the reverse DNS lookup either.

This will render our technique completely useless. However, the technique in [Gru98] will still have a chance of working.

6 LOAD DETECTION

The last technique we present for remote sniffer detection is a more general technique with fewer assumptions. The only assumption made is that the suspected sniffing host runs a service, such as FTP. The load detection technique can be used to detect remote sniffers across a router. However, this technique is by far the trickiest to implement.

6.1 Sniffer Interaction on Machine Resources

The load detection technique works by noting that NIC interrupts to the operating system degrades system performance. When sniffers are running, the NIC is put in to promiscuous mode as previously discussed. Since the NIC is in promiscuous mode, all Ethernet traffic will generate hardware interrupts which will cause the Ethernet driver code to execute. Furthermore, with a sniffer running all captured packets must be passed to the user program running the sniffer. Crossing the kernel boundary is widely known to be relatively expensive. Thus, under heavy traffic, a sniffer can heavily degrade performance on a promiscuous host.

The load technique, then, is to somehow remotely measure the machine load when there is heavy traffic on the segment. The obvious way to do this is to take a measurement of response time from the machine that is suspected of running a sniffer. However, how this measurement is taken is the major difficulty in implementing this technique.

6.2 Implementation

Figure 5 shows roughly how the load technique works. In the load technique, two measurements of response time is taken. One measurement is taken to determine the response time of the machine without heavy network traffic. Another measurement is taken to determine the response time of the machine with heavy traffic. The response time of these two measurements are compared to determine whether a sniffer is running on the host or not.

In our implementation, each measurement runs through several trials, and the response time is averaged. In the base case measurement, the measurement taken without heavy network traffic, the response time of the FTP service running on the suspected host is taken. The response time is determined by the amount of time to do a `connect` and a `recvmesg` from the FTP service, thus, SYN and ACK packets are sent by the suspected host. In the current implementation, 20 such trials are performed, with each trial a small 5 second delay is imposed. Then, a second measurement is taken. This measurement is taken by measuring the response time of the FTP service with some traffic interspersed in between the `connect` and `recvmesg` call.[5] The interspersed traffic is a series of fake FTP connections targeted to some other host. Thus, hosts which are not in promiscuous mode will completely ignore this traffic while hosts that are in promiscuous mode will not ignore this traffic. The response time of these two measurements are compared to determine if a host is in promiscuous mode.

The implementation described so far is quite simple. However, it is tricky to implement because there were so many pitfalls to avoid while designing the implementation. The footnote regarding how packets should be sent is just one possible pitfall. Another pitfall is determining what should be used to determine the response time. The obvious choice is to use the "ping" packet. However, this turns out to be a very bad choice. The ICMP Echo Request packet handler in most TCP/IP stacks handle the ICMP Echo Request packet as soon as it is received from the network. That is, when a ICMP Echo Request packet is received by the TCP/IP stack, it is processed right away by returning the ICMP Echo Reply packet. Thus, no where in this path does

[5] A small pitfall should be avoided when attempting to implement this. The `connect` system call should be done nonblocking, so that the operating system returns right away. Then, some fake FTP connection traffic to the network segment, destined to some other machine, should be sent nonblocking as well. Finally, the `recvmsg` system call should be done blocking, of course. If the `connect` was not done in nonblocking mode, our timing results would be incorrect since we were then unable to pipeline the fake traffic.

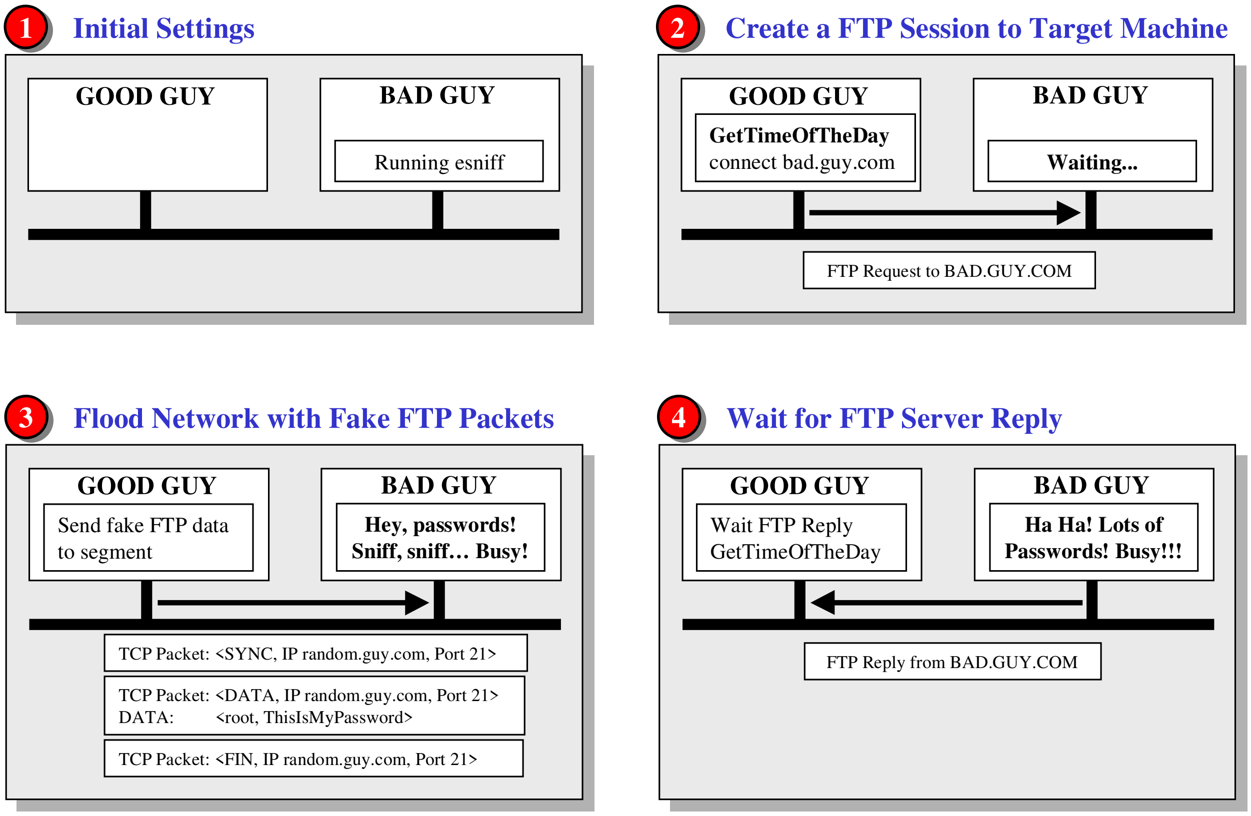

Figure 5: **Diagram for the load detection technique** *This figure shows how the load detection technique works.*

the operating system scheduler run to cause a context switch. In effect, the ICMP Echo Request and Reply is atomic. Therefore, ping packets cannot be used to measure load since user programs are never executed during the processing of ping packets. Likewise, a single `connect` system call is insufficient to measure response time. This is because when a user program binds to a port to accept connections, if a SYN packet is received by the operating system to connect to the port, the operating system replies with the SYN and ACK right away. Therefore, user programs, again, do not have a chance to run. Thus, we were forced to choose a service such as FTP and measure not only the time to establish a TCP connection, but also the time to actually receive a piece of data generated by the user program (in this case, a hello message from the FTP server). In this way, the scheduler is allowed to run and context switches are allowed to occur, so we were able to measure the load of the machine.

6.3 Analysis

Table 1 and figure 6 shows the statistical summary of the remote machine load detection experiments. The experiments were carried out on three different systems: a Pentium 200MHz system running Linux v2.0.35 (*Linux 200*), a Pentium II 300MHz system running Linux v2.0.35 (*Linux 300*), and a Pentium II 300MHz system running FreeBSD v2.2.7 (*FreeBSD 300*). Four different experiments were performed on each system and the round-trip time of a FTP connection to the target system was recorded. Each experiment was repeated twenty times to reduce chance error in sampling. The four experiments are:

- *no sniffer is running* on the target system and *no fake TCP packet was sent* (*sniff off, fake off*);
- *no sniffer is running* on the target system and *fake TCP packets were sent* (*sniff off, fake on*);

	Testing Mode	Average RTT (*msec*)	Precentage Change	Standard Deviation
Linux 200	Sniff off, Fake off	45.34		10.44
	Sniff off, Fake on	46.32	2.16%	8.52
	Sniff on, Fake off	49.99		12.54
	Sniff on, Fake on	59.65	19.32%	11.84
Linux 300	Sniff off, Fake off	32.39		3.43
	Sniff off, Fake on	33.18	2.43%	3.61
	Sniff on, Fake off	34.46		2.67
	Sniff on, Fake on	42.42	23.09%	3.37
FreeBSD 300	Sniff off, Fake off	12.42		0.08
	Sniff off, Fake on	15.38	23.81%	2.18
	Sniff on, Fake off	12.49		0.08
	Sniff on, Fake on	20.34	62.79%	2.82

Table 1: **Machine Load Statistics** *This table shows the statistical summary of the remote machine load detection experiments. Three target systems were tested: Pentium 200MHz running Linux 2.0.35 (***Linux 200***), Pentium II 300Mhz running Linux 2.0.35 (***Linux 300***), and Pentium II 300Mhz running FreeBSD 2.2.7 (***FreeBSD 300***). The tests were carried out in four different scenarios: sniffer was not running and no fake TCP packet was sent (***Sniff off, Fake off***), sniffer was not running and fake TCP packets were sent (***Sniff off, Fake on***), sniffer was running and no fake TCP packet was sent (***Sniff on, Fake off***), and sniffer was running and fake TCP packets were sent (***Sniff on, Fake on***). The average round-trip time of connecting to the FTP service on the remote target is calculated (***Average RTT***).* **Percentage change** *shows the percentage increase in average round-trip time with* **fake off** *shifting to* **fake on**, *and the* **standard deviation** *measures the spread of each data set.*

- *sniffer is running* on the target system and *no fake TCP packet was sent* (*sniff on, fake off*);
- *sniffer is running* on the target system and *fake TCP packets were sent* (*sniff on, fake on*).

On *Linux* systems, a total of 6 fake TCP packets (2 fake FTP connections) were sent; whereas on *FreeBSD 300*, a total of 48 fake TCP packets (16 fake FTP connections) were sent. This is due to the fact we need to send more fake TCP packets in order to show a significant increase in response time of the FTP connection to the target host which has a more optimized network layer.[6]

The results show that on both *Linux* systems, there is roughly a 2 percent increase in response time of the FTP connection on a sniffer-free machine when fake TCP packets are sent. However, there is roughly a 20 percent increase in response time when there is a sniffer running. This significant increase in response time clearly distinguishes a sniffer-free system from a system that has a sniffer running.

Surprisingly, the fake TCP packets *do* increase the response time of the FTP connection by 20 percent even the sniffer is not running on the *FreeBSD 300* system. This increase can be explained by the higher network delay introduced by injecting more fake TCP packets onto the network. However, when the sniffer is running on the target host, it increases the response time by another 40 percent, a total of 60 percent increase.

From the percentage difference of the average round-trip time between *fake off*, *fake on*, we can see that there is a significant increase in round-trip time of the FTP connection to the target host when there is a sniffer running. We also notice that the round-trip time is higher with *fake on* compare to *fake off*, no matter whether the sniffer is running or not. In order to claim that this technique is useful, we need to show that the average of the sample round-trip time is statistically different enough to make the judgement on whether the sniffer is running. We employed z-statistics [FPP+91] on the sample averages to

[6]The Berkeley Packet Filter (BPF) [MJ93] module in FreeBSD is more optimized than that of the simple filter in Linux. In BPF, multiple packets can be returned on one system call, whereas in Linux, each packet is returned by a single system call. Thus, the FreeBSD tests require more packets to be sent to trigger more system calls.

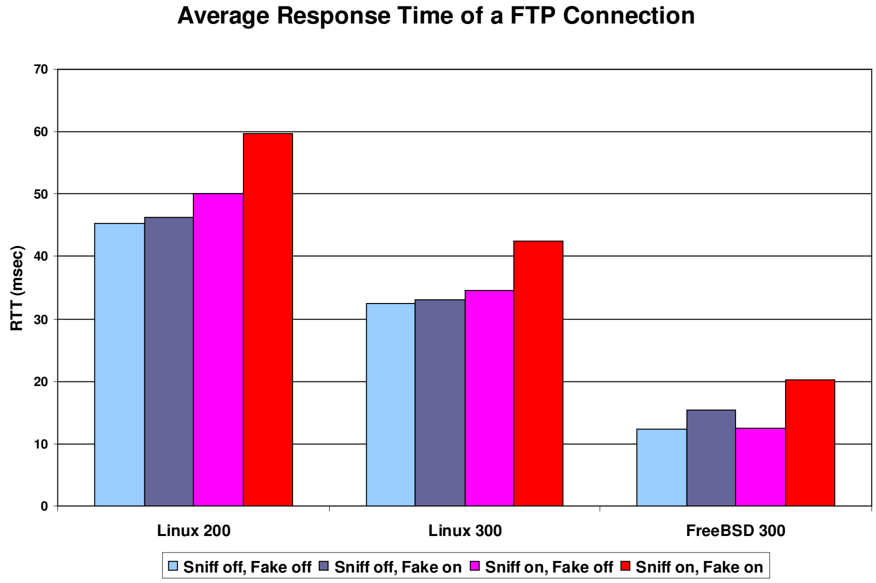

Figure 6: **Average round-trip time of FTP connection** *This figure shows the average round-trip time of a FTP connection for four different experiments carried out on three different systems.*

answer the following questions:

- If the sniffer is not running, will the fake TCP packets affect the average round-trip time of the FTP connection? (Sniff off, Fake off) vs. (Sniff off, Fake on)
- If the sniffer is running, will the fake TCP packets increase the average round-trip time of the FTP connection? (Sniff on, Fake off) vs. (Sniff on, Fake on)
- If the fake TCP packets were sent, will the average round-trip time of the FTP connection affected by a running sniffer? (Sniff off, Fake on) vs. (Sniff on, Fake on)

Specifically, we use:

$$z = \frac{(Average_A - Average_B)}{\sqrt{(\frac{Standard\ Deviation_A}{\sqrt{Size_A}})^2 + (\frac{Standard\ Deviation_B}{\sqrt{Size_B}})^2}}$$

where,

- z is the z-statistics;
- $Average_A$ is the sample average of data set A;
- $Average_B$ is the sample average of data set B;
- $Standard\ Deviation_A$ is the sample standard deviation of data set A;
- $Standard\ Deviation_B$ is the sample standard deviation of data set B;
- $Size_A$ is the size of the sample data set A;
- $Size_B$ is the size of the sample data set B.

Table 2 shows the calculated z-statistics for three different set of sample averages on the systems we tested. On *Linux 200* system, the difference in round-trip time is not significant when the sniffer is not running ($z = 0.3252\ SE$). However, when the sniffer is running, the average round-trip time of FTP connection with sending fake TCP packets is differentiable from without sending the fake TCP packet ($z = 2.5049\ SE$). This difference is due to the fake TCP packets triggering the sniffer to become active on the target system. This result is confirmed with the z-statistics for the third scenario, (Sniff off, Fake on) vs. (Sniff on, Fake on). The z-statistics ($z = 4.0868 SE$) prove that the difference in the sample averages under these two tests is unlikely to be caused by chance error, which implies that a running sniffer *does* increase the response time of a FTP connection when fake TCP packets are sent.

	Comparison	z-statistics
Linux 200	(Sniff off, Fake off), (Sniff off, Fake on)	0.3252 SE
	(Sniff on, Fake off), (Sniff on, Fake on)	2.5049 SE
	(Sniff off, Fake on), (Sniff on, Fake on)	4.0868 SE
Linux 300	(Sniff off, Fake off), (Sniff off, Fake on)	0.7095 SE
	(Sniff on, Fake off), (Sniff on, Fake on)	8.2796 SE
	(Sniff off, Fake on), (Sniff on, Fake on)	8.3673 SE
FreeBSD 300	(Sniff off, Fake off), (Sniff off, Fake on)	6.0693 SE
	(Sniff on, Fake off), (Sniff on, Fake on)	12.4445 SE
	(Sniff off, Fake on), (Sniff on, Fake on)	12.2924 SE

Table 2: **z-Statistics tests** *This table shows the results of one-tail z-statistics on the average round-trip time of the experiments.*

Like *Linux 200* system experiments, the calculated z-statistics on *Linux 300* system strongly shows that the difference in the sample average round-trip time for sending fake TCP packets when the sniffer running is significantly higher and distinguishable from the average round-trip time of not sending any fake TCP packets ($z = 8.2796\ SE$). Together with the insignificant difference ($z = 0.7095\ SE$) under the first scenario, (Sniff off, Fake off) vs. (Sniff off, Fake on), the experiments help us to reconfirm that this technique can be used to distinguish a target host with sniffer is running from a target host without it.

On *FreeBSD 300*, however, the z-statistics disapprove the claim that the average round-trip time is alike when the sniffer is not running ($z = 6.0693\ SE$). This difference in average round-trip time is due to the fact that we need to send eight times more fake TCP packets, a total of 48 fake TCP packets, in order to notice a significant difference in the response time of the FTP connections when the sniffer is running on a highly optimized system. By sending more packets, this introduces network delay which contributed to a significant increase in response time of the FTP connection even though the sniffer is not running. On the other hand, the z-statistics shows that there is an even stronger difference in response time of the FTP connection when the fake TCP packets were sent and a sniffer is running on the target host ($z = 12.4445\ SE$). This is explained by the workload introduced to the target host when the sniffer is running in addition to the network delay introduced. This result is again reconfirmed by calculating the z-statistics on experiments with fake TCP packets sent to a running sniffer system against a sniffer-free system ($z = 12.2924\ SE$).

As shown in table 1, the change in response time for a *FreeBSD* system is very different from the *Linux* systems. We cannot concluded that a remote system has a sniffer running by just having a 20 percent increase in response time (it might be running FreeBSD). However, by knowing the characteristics of the remote system, together with the z-statistics, we can argue that a system running *Linux* with 20 percent increase in response time, and a system running *FreeBSD* with 60 percent increase response time and more than 10 SE different in z-statistics will have sniffers running.

6.4 Results

As shown in the analysis section, the load detection technique is quite effective. We were able to detect sniffers with pretty good accuracy. In fact, we were able to determine interesting machine characteristics, such as the operating system type and machine performance by measuring the load of the machine.

However, there are a few shortcomings in this technique. The load detection technique works well only if the network is relatively quiet. That is, the network should have very little traffic when the tool is run. When the network is already on heavy traffic, it is hard to accurately determine the machine load based on the response time sine the response time of the machine may be slowed due to Ethernet contention. Nonetheless, this is probably a reasonable restriction since this tool can be run late at night, when there should be very little traffic, but the sniffer is still at work. Another problem with this technique, however, is that if the suspected machine is already under heavy load, it will be hard to detect a load at all since the two average response times will not differ much.

7 DISCUSSION

The three techniques introduced in this paper are quite effective in aiding a system administrator in determining whether a sniffer is running on a network. Each of the techniques have unique constraints. In this section, we will examine the differences among the three techniques.

7.1 Accurracy

In terms of accuracy, all techniques are quite effective. The MAC detection technique is 100% accurate if all host operating systems exhibit the implementation anomaly in the network stack. However, we have shown that there are operating systems which do not have this anomaly, so this technique is not completely accurate. The DNS detection technique is currently 100% accurate since all password sniffers perform the reverse DNS lookup. The load technique, as shown in the analysis section, is quite accurate, but we do not have enough data yet to quanitfy the accuracy rate. More work will need to be done in this area. Certainly, the load technique is not 100% accurate.

7.2 Resistence

In terms of resistence, that is, how well the technique can be thwarted, the load technique seems to be the most resistant. To defeat this technique, all services will have to be turned off to prevent measurement. However, by doing so, the tool can be modified to signal a warning that for some reason the service is turned off. The MAC technique can be defeated on machines on which this technique works by patching the kernel. This is not hard to do, but would typically require a reboot of the machine. Depending on how the machine is used, a random reboot might set off an alarm that something suspicious is happening. The DNS technique is the weakest in terms of resistence. Password sniffing tools can be modified to disable the reverse DNS lookup. In fact, it is expected that in a short period of time after this paper is released, most password sniffers will be modifed to disable the reverse DNS lookup, and hackers will educate each other on why reverse DNS lookup should not be performed.

7.3 Detection Across a Router

The MAC detection technique can only be used on the same network segment as the suspected password sniffing host. However, both the DNS detection technique and the load detection technique can be run on a network that is different from the network segment that the suspected password sniffing host is running on. That is, both of these techniques can be done across a router. However, whereas the DNS detection technique maintains the same accuracy level across multiple routers, the more routers that are in the way of the load detection tool and the suspect host, the less accurate the detection since the response time can be delayed by intermediate routers.

7.4 Ease of Use

Both the MAC detection and load detection techniques are easy to use. However, the DNS detection technique is a bit harder to use because it requires some administrative setup. Specifically, the DNS servers need to be changed so that the trigger IP address is registered to be served by the machine that runs the DNS detection tool.

8 FUTURE WORK

The work presented in this paper is by no means complete. More analysis will need to be done, and some of the tools need to be reworked. Furthermore, the tools need to be integrated in to a package.

For the MAC detection and load detection techniques, more operating systems need to be tested. We have only perform testing on two different operating systems, Linux and FreeBSD. Much more work needs to be done on checking the results on different operating systems.

For the load detection technique, not only do we need to test on more operating systems, but we also need to test on more machines. The performance of various machines may exhibit different behavior. We have only tested on two different machines, a Pentium 200MHz and a Pentium II 300MHz. Much more testing on different hardware platforms and configurations will be necessary.

The timing and number of trials we used in the DNS detection and load detection techniques were mostly obtained empirically. A complete study of these techniques should include work on minimizing the number of trials and the amount of time necessary for the implementation. Although both techniques currently run relatively quickly (roughly 3 minutes per network on the DNS detection tool and about 1 minute per host for the load detection technique), it would be interesting to see how far we can optimize the number of trials and the amount of delay time.

For the DNS detection technique, we need to reimplement the tool such that a DNS reply TTL of 0 is returned. As previously mentioned, this will eliminate the problem of receiving duplicate DNS lookup request by the same host due to time out. This will reduce the amount of false positives.

For the load detection technique we need more data on how much accurate the technique is when more intermediate routers are introduced. Our tests are only applied with at most two intermediate routers with in a local campus. While this proves that the technique works and is feasible in corporate environments, it does not show how well the technique works from slow connections and from far remote locations.

Currently, the tools are relatively primitive. These tools still require manual execution, although the techniques can be use in an automated tool. In fact, the DNS and load detection tools do not return an answer such as "a sniffer is running", but rather numerical results are returned with a person will have to analyze. However, it is not hard to extend both tools to return such a simple answer. For the DNS tool, the tool needs to be extended so that it can check if the fake traffic actually connects to a real host and decrement the number of DNS requests accordingly. Then, it can print the results if the resulting number of requests is greater than zero. For the load detection tool, the statistical analysis we presented in this paper can be programmed in to the tool. The tool can perform the calculations and determine whether the sniffer is running or not and output the result.

Lastly, these tools, or rather, the techniques presented, should be integrated in to a sniffer detection suite. It would be more user friendly if a graphical frontend is created for these tools. Furthermore, making these tools automated, so that, for example, these tools can automatically run at night, when legitimate uses of `tcpdump` should not occur, will be very helpful.

9 CONCLUSION

In the past, it was widely believed that sniffer detection is hard or impossible, or the constraints required for a workable sniffer detector is to severe to be of any use. In this paper, we have presented three techniques which are effective in remotely detecting sniffers running on a network. These techniques prove that not only is sniffer detection possible, but it is practical and can be highly accurate.

Our experience from developing and implementing these techniques suggest that before labelling a task impossible, one must carefully study the task at hand. There were many previous claims on the infeasibility of creating a practical sniffer detection. One claim is that there is no way to detect all sniffers because some sniffers are true passive devices that listen on the network. Although this claim is true, there are numerous remote hacker break-ins that are aided by password sniffing programming running on normal hosts. The fallacy in this claim is that one does not need to handle all possibilities; the common cases are good enough. Another claim is that sniffers are completely passive. This claim is not true at all. With careful examiniation of sniffer programs, we were able to determine that sniffers can behave actively. We found that they all perform reverse DNS lookups. The lesson to be learned is that careful scrutiny must be applied. Lastly, it is claimed that remote measurement of machine load is impossible and too inaccurate. Although we found that to come up with the right steps to measure load is difficult, we have shown that it can be done. We found that one must have a good understanding of the interaction with in the machine in order to come up with a good technique for remote load measurement. For the load detection technique, one really needs to fully understand how the NIC and the operating system interact, as well as how the network stack interact with the scheduler and how the scheduler works to schedule user programs. Furthermore, care must be taken to understand how the network layer handles connection requests and how data is received. With a deep understanding of the complete path taken, we were able to come up with the right technique to measure machine load remotely. Thus, sometimes things are said to be impossible only because no one has really tried hard enough to understand the task at hand.

Unfortunately, these techniques can be rendered useless quite easily. Therefore, sniffer detection systems are just stop gap measures. For the truly security conscious, real authentication and encryption should be used. No clear text passwords should be sent anywhere on the network. Secure services such as `ssh` [YKS+98] are now widely available. These secure services should be used and the insecure services they replaced should be completely shut off. Only in this way is better security obtained.

References

[FPP+91] Freedman, Pisani, Purves and Adhikari, "Statistics - Second Edition", (pp. 457-459), W.W. Norton & Company, Inc., 1991.

[Gru98] Grundschober, S., "Sniffer Detector Report", Global Security Analysis Lab., Zurich Research Laboratory, IBM Research Divison, June 1998.

[Hor84] Hornig, C., "A Standard for the Transmission of IP Datagrams over Ethernet Networks", RFC-894, Symbolics Cambridge Research Center, April 1984.

[JLM89] Jacobson, V., Leres, C., and McCanne S., "The Tcpdump Manual Page", Lawrence Berkeley Laboratory, Berkeley, June 1989.

[Kan91] Kantor, B., "BSD Rlogin", RFC-1258, Universify of California San Diego, September 1991.

[MJ93] McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Proceedings of the 1993 Winter USENIX Technical Conference, January 1993.

[Moc87-1] Mockapetris, P. V., "Domain Names - Concepts and Facilities", RFC-1034, USC/Information Sciences Institute, November 1987.

[Moc87-2] Mockapetris, P. V., "Domain Names - Implementation and Specification." RFC-1035, USC/Information Sciences Institute, November 1987.

[MR94] Myers, J., and Rose, M., "Post Office Protocol - Version 3", RFC-1725, Carnegie Mellon/Rover Beach Consulting, Inc., November 1994.

[Plu82] Plummer, David C., "An Ethernet Address Resolution Protocol – Converting Network Protocol Addresses to 48 bit Ethernet Address for Transmission on Ethernet Hardware", RFC-826, November 1982.

[Pos81-1] Postel, J., "Internet Protocol", RFC-791, USC/Information Sciences Institute, September 1981.

[Pos81-2] Postel, J., "Transmission Control Protocol", RFC-793, USC/Information Sciences Institute, September 1981.

[Pos94] Postel, J., "Internet Control Message Protocol", RFC-792, USC/Information Sciences Institute, September 1981.

[PR83] Postel, J., and Reynolds, J., "Telnet Protocol Specification", RFC-854, USC/Information Sciences Institute, May 1983.

[PR85] Postel, J., and Reynolds, J., "File Transfer Protocol", RFC-959, USC/Information Sciences Institute, October 1985.

[YKS+98] Ylonen, T., Kivinen, T., Saarinen, M., Rinne, T., and Lehtinen, S., "SSH Protocol Architecture", Network Working Group, IETF, Auguest 1998. http://search.ietf.org/internet-drafts/draft-ietf-secsh-architecture-02.txt

[Uni97] "Firewall Mailing List Archive 1997", Unix, Internet, and Security, 1997. http://www.netsys.com/firewall.html

[DIX80] "The Ethernet - A Local Area Network, Version 1.0", Digital Equipment Corporation, Intel Corporation, Xerox Corporation, September 1980.