The Development and Analysis of Intrusion Detection Algorithms

by

Seth E. Webster

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

© Seth E. Webster, MCMXCVIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .....
Department of Electrical Engineering and Computer Science
May 27, 1998

Certified by .....
Dr. Richard P. Lippmann
Senior Staff MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Development and Analysis of Intrusion Detection Algorithms

by

Seth E. Webster

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents three new low-complexity intrusion detection algorithms tested on sniffing data from 80,000 real Internet sessions. A Password Guessing Detector scans telnet connections containing only failed logins and identifies password guessing attacks based on the number of connection between each host pair and the usernames and passwords tried. By extracting the plaintext password, this algorithm is able to run much faster than conventional cracking programs, which must encrypt each guess. A neural network based keyword weighting system substantially improves the performance of a baseline intrusion detection system. It uses counts of forty keywords, also found in the sniffing data, to identify attacks in telnet connections. Finally, a real-time Shell Tracker monitors BSM audit data and finds users who illegally become root and start a command shell regardless of the attack or back door used. Two intrusion prevention algorithms, a Banner Finder and a Password Checker, were also developed. The Banner Finder uses a binary decision tree and five keyword counts to identify the presence of a valid warning banner in telnet sessions. The Password Checker extracts usernames and passwords from network sniffing data and scores the passwords based on how succeptable they would be to a password guessing attack.

Thesis Supervisor: Dr. Richard P. Lippmann
Title: Senior Staff MIT Lincoln Laboratory

Acknowledgments

First and foremost, I would like to thank Rich Lippmann, my supervisor, for continually pointing me in the right direction and keeping track of the big picture. I would also like to thank Dan Wsychograd for the extensive work he did analyzing and classifying the data. Dan Weber, and his attack scripts, were invaluable for testing many of the algorithms. Rich, Dan Wsychograd, Dan Weber, Sam Gorton, and all the other Lincoln Lab folks working on the intrusion detection project provided hours advice, ideas, discussion, and feedback which, more than once, kept me from heading down the wrong path. Finally, I would like to thank Teresa Lunt at DARPA and the Department of Defense.

Contents

1 Introduction **9**

2 Background **11**

2.1 Overview of Computer Intrusions 11

2.2 Current Intrusion Detection Research 13

2.2.1 Anomaly Detection 13

2.2.2 Misuse Detection 16

2.2.3 Intrusion Prevention 17

2.3 Input Data Sources for Intrusion Detection 17

2.3.1 Audit Data 18

2.3.1.1 Overview 18

2.3.1.2 Solaris Basic Security Module 19

2.3.2 Sniffer Data 21

2.3.2.1 Overview 21

2.3.2.2 TCP Sessions 22

2.3.2.3 Transcript Creation 23

2.3.2.4 Attacks in Data 25

2.3.2.5 Problems With Transcripts 28

2.4 The Baseline System 29

2.5 ROC Curves and Purity 33

3 Password Guessing Detector **36**

3.1 Dictionary and Doorknob Rattling Attacks 37

3.2 The Password Guessing Detector 38
3.3 Password Guessing Detector Results 40

4 Keyword Reweighting System 43
4.1 The Neural Network 43
4.2 Results 44

5 Bottleneck Verification Prototype 47
5.1 Bottleneck Verification Overview 47
5.2 The Shell Tracking Algorithm 48

6 Intrusion Prevention Algorithms 51
6.1 Warning Banner Finder 51
6.1.1 Description of a Valid Warning Banner 51
6.1.2 The Warning Banner Finder Algorithm 52
6.1.3 Warning Banner Finder Results 53
6.2 Password Checker 54
6.2.1 Password Extraction Algorithm 55
6.2.2 Password Checking Algorithm 58
6.2.3 Password Checker Results 59

7 Conclusions 60
7.1 The Password Guessing Detector 60
7.2 New Keyword Weighting System 61
7.3 Shell Tracking Algorithm 62
7.4 Intrusion Prevention Algorithms 62

8 Future Work 63
8.1 Password Guessing Detector 63
8.2 New Keyword Weighting System 64
8.3 Bottleneck Verification Algorithm 64
8.4 Password Checker 65

A Tcptrans Transcript Generation Utility **67**

A.1 Current Functionality 67

A.2 Future Plans 68

Bibliography **73**

List of Figures

2-1 The Different Sources of Data For Intrusion Detection Systems 18
2-2 Example of Multiple Connections Between Two Hosts 23
2-3 Init File For a Transcript Generated by the Baseline System 24
2-4 Dest File For a Transcript Generated by the Baseline System 25
2-5 Breakdown of Telnet Transcripts 26
2-6 Examples of Character Doubling in Transcripts 29
2-7 Normal Operation of the Baseline System 30
2-8 Sample ROC Plot 34

3-1 Sample Output of the Password Guessing Detector for a Series of Suspicious Connections 42

4-1 Neural Network Used to Score Connection 45
4-2 Performance of Baseline System and Neural Network 46

5-1 Example of a Bottleneck Between Two Groups of States 48

6-1 Decision Tree for Warning Banner Finder 54
6-2 Logic Used to Locate a Password 57

A-1 The Tcptrans Transcript Generation Utility 69
A-2 Sample Init File Generated by the Tcptrans Utility 69
A-3 Sample Dest File Generated by the Tcptrans Utility 70
A-4 Sample Summary File Generated by the Tcptrans Utility 70

List of Tables

2.1 Different Types of Intrusion Detection Systems 14

2.2 Basic Security Module Token Types 20

2.3 Day/Weight Pairs 32

2.4 Sample ROC Data 34

3.1 Names of Commonly Attacked Accounts 38

3.2 Password Guessing Algorithm Results 41

6.1 Permutations Used By Password Checker 59

A.1 Currently Supported Options for Tcptrans 71

Chapter 1

Introduction

More computers are connected to the Internet today than ever before. Unfortunately, system security has not kept up with the increase in connectivity. The expansion of the Internet has thus enabled a corresponding rise in computer crime. With the realization of electronic banking and commerce over the Internet, the stakes are even higher. Many businesses would experience crippling financial losses if the information on their computers was lost or stolen.

Unfortunately, it is unlikely that systems in the near future will be more secure than those of today. Operating systems and applications continue grown, becoming not only more powerful, but also more complex. This complexity makes these systems more likely to contain bugs, some of which will be exploitable by attackers to gain access to the system and its data. Thus, a relatively serious problem exists and it is growing worse. As a greater volume of sensitive data is accessible over the Internet, the potential rewards hackers can receive also increases. External monitoring of systems to detect attacks is thus becoming increasingly important.

Intrusion detection systems are systems designed to monitor a computer or network of computers for malicious activity. They offer some important benefits. One benefit is the deterrent value of making it known to users that the systems they are on are being monitored. This warning may be enough to cause "joy riders" (high school attackers who are breaking into systems simply for fun) to look for potential victims elsewhere.

A second benefit is that even a network which is secure against all known threats may be vulnerable to as-yet-undiscovered attacks. Without monitoring, the first sign a company may see of an attacker is system failures or corrupted data. A good intrusion detection system would report to system administrators the presence of an attack soon after it happens (or as it is happening in the case of real-time systems). Thus, attacks can possibly be shut down before any damage occurs.

Log files kept by an intrusion detection system often contain a record of the attack itself as well as the actions the intruder took once he/she broke in. This information can be used to close a security hole before it is exploited again, to undo any changes made by the attacker, and even as evidence to confront or prosecute a malicious employee outside attacker.

The focus of this thesis is on intrusion detection in UNIX environments. It presents some new intrusion detection algorithms as well as lessons learned while developing those algorithms. The next chapter contains some background information, including a description of various intrusion techniques, an overview of current intrusion detection research, the data sources used as inputs to the new intrusion detection algorithms, a description of the baseline system which the new algorithms are compared to, and an explanation of the metrics used to compare the different algorithms. Chapter 3 gives an overview of the Password Guessing Detector. Chapter 4 presents a system which uses a neural network to improve the performance of the baseline system. Chapter 5 describes the bottleneck verification approach and the Shell Tracker, a prototype system which uses this approach to find illegal transitions to root. Chapter 6 presents two intrusion prevention algorithms which identifies potential security problems before they are exploited. Chapter 7 is a discussions of the conclusions drawn from this work about the algorithms themselves and about designing intrusion detection algorithms in general. Finally, chapter 8 lists some ideas for future improvements to the various algorithms presented in this thesis.

Chapter 2

Background

2.1 Overview of Computer Intrusions

For the purpose of this thesis, an intrusion is defined as an interaction with a computer system in which, at any time during that interaction, an individual operates with privileges higher than he/she should legitimately have. Under a UNIX operating system, this could include a user exploiting a bug in an operating system to gain root privileges (the highest level of permissions and exempt from all of the operating system's security policies) as well as an outsider logging in to a legitimate user's account using a stolen password. Intrusions are a subset of computer attacks, where an attack is defined as any malicious activity directed towards a computer system or the services it provides. Some examples of computer attacks are viruses, physical damage, and denial-of-service attacks. Since this thesis focuses on intrusions and intrusion detection, the terms "intrusion" and "attack" will often be used interchangeably to mean "intrusion" as it is defined above.

There are many ways for an intruder to gain illegal access to a system. A few examples are:

Software bug Typically, either the operating system or an application running as root is subverted into running arbitrary code of the attacker's choosing. What code the attacker will have the operating system or application run varies, but

some common actions are to return a command shell running as root or to add a user to the system with a specific password and root permissions.

System misconfiguration This category includes attacks exploiting accounts with no passwords or files with the wrong permissions. For example, the default setup for some systems includes an account named "guest" which will accept any password. In addition, an application, such as a web server, could be misconfigured to allow outside users access to sensitive areas of the file system, like the password file.

Social engineering Social engineering refers to attacks in which people with legitimate access to the system are fooled into giving access to an attacker. Two examples are calling company personnel while pretending to be an administrator and asking for a password, or mailing a company a piece of software, apparently from the vendor, which claims to be an upgrade. This software will actually have a back door giving an attacker access to the system once installed.

Password sniffing This attack generally involves a machine on the local network which the attacker has already broken into. This machine is used to monitor all network traffic and capture usernames and passwords as other users log in remotely.

Password guessing/cracking A brute force strategy in which an attacker attempts to guess a user's password. The attacker can either repeatedly try to log in as the user with different passwords (referred to as a dictionary attack), or copy the password file to their own machine and run a password cracking program (such as CRACK[14]) on it.

These categories in this list are not mutually exclusive, and the list is not meant to be exhaustive. It is meant to show the variety of attacks a secure system must defend against (or a good intrusion detection system must be able to detect).

2.2 Current Intrusion Detection Research

Intrusion detection is the field of research concerned with building systems to detect and/or defend against intrusions. Most systems focus on intrusions as they are defined in the previous section, however some systems also look for other kinds of attacks, such as viruses or denial-of-service attacks. Generally speaking, current intrusion detection systems can be classified as either anomaly detectors or misuse detectors (or both). Anomaly detectors model normal behavior and try to detect attacks by looking for unusual, though not necessarily illegal, usage patterns. Misuse detection systems look for illegal behavior with respect to the operating guidelines of a system. Unlike anomaly detection systems which can only model normal behavior, misuse detection systems can model both legal behavior and illegal behavior (attacks) directly. The difference is that attacks are relatively well defined (especially known attacks) and can be looked for directly while user behavior can only be defined as anomalous when compared to normal behavior for that user. Within the categories of anomaly detection and misuse detection, most current systems either use an expert system with a fixed set of rules or some statistical or learning algorithms to model the target behavior. Table 2.1 shows where some current systems, including the Shell Tracker and Keyword Reweighting System (presented later in this thesis), fit into the above classification. The columns indicate the type of behavior being modeled and the rows give the mechanism used to generate the models. Systems which model more than one type of behavior or use a combination of techniques to represent the models are listed in all appropriate places in the table.

The following sections describe both the systems listed in Table 2.1 as well as intrusion prevention systems, which do not fit well in the above classification.

2.2.1 Anomaly Detection

Anomaly detection systems attempt to detect intrusions by building models of normal system activity and flagging any actions which deviate significantly from those models. These models may describe the actions of each user individually, or groups of users

Modeling Technique	Type of Behavior Modeled By System: Normal Behavior	Legal Behavior	Illegal Behavior
Statistical or Learning Algorithms	Haystack NIDES Neural Network Systems	Keyword Reweighting system	Keyword Reweighting System
Rule Based Expert System	GrIDS	Shell Tracker Haystack Specificaton Based	Model-Based Intrusion Detection NSM STAT NIDES

Table 2.1: Different Types of Intrusion Detection Systems

(such as administrators, software developers, secretaries, etc.).

NIDES (Next-Generation Intrusion Detection Expert System) has both an anomoly detection component and a misuse detection component. The anomaly detector builds models of individual users and then compares those models to current behavior using a statistical approach. It constructs short- and long-term profiles for each user from a variety of measures extracted from audit data (described in section 2.3.1) and periodically monitors statistics for the two profiles to determine if current activity is within normal limits. Both the short and long term profiles are built by combining all recent audit records such that newer records have more influence on the profile than older records. The half-life of a profile describes how much weight the contribution of a given record has based on the age of that record. For example, a half-life of 200 records indicates that the 200^{th} record contributes only half as much as the current record and the 400^{th} record only 1/4 as much. The half-life of the short term profile is usually described in terms of audit records while the half-life of the long term profile is usually described in days. The misuse detection component detects known attacks by scanning the audit data for sequences of events described by a rule in the rule base. Each rule has a rulebase of facts which are updated based on the audit records

seen by the system. A rule can fire when it has all the necessary facts and all other external conditions for that rule are met [10].

Haystack, another combined anomaly detection/misuse detection system, uses both user and group models. It has static templates designed by the security officer which are used as initial profiles for new users based on the types of actions these users are expected to perform. Over time, a user's initial profile is modified to reflect his or her actual behavior. These profiles can be periodically monitored by the security officer to detect attackers attempting to train the system into considering attacks as normal behavior. The audit data used to train the user profiles is also scanned for patterns corresponding to known attacks [17].

GrIDS (Graph-Based Intrusion Detection System) differs from the other anomaly detection systems discussed so far in that, instead of modeling the activity of an individual user or host, it monitors the activity of an entire network. User-specified rules describe how to construct different graphs and when to raise an alarm based on the graph structure [15].

Finally, neural networks have also been used in anomaly detection systems. Some preliminary results have been obtained training a neural network to predict the next command a user will issue based on previous commands [2]. In addition, a Neural Network Intrusion Detector (NNID) uses a neural network to learn the behavior of a particular user and distinguish that user from an attacker logging in to the same account [16].

Anomaly detection systems have the advantage of being able to detect masquerading and novel attacks. For example, it would be very difficult for misuse detection systems to catch an intruder logging in with a stolen password and reading or modifying files owned by that account, since the login and all commands issued would be perfectly legitimate. An anomaly detection system, however, may be able to notice the difference between the actions the attacker takes once logged in and the expected actions based on a model of that user. Likewise, a system which searches for known attack signatures will miss any new attack which was not explicitly included in its database. If the attack or the attacker's actions are different enough from normal

activity, an anomaly detection system will catch them.

Their main disadvantage, however, is that an anomaly detection system will only work if normal user or system activity is sufficiently stable over time and does not overlap with attacker activity. A user with very regular habits will be much easier to model and intruders using such an account will typically exhibit activity which deviates significantly from normal. However, if users have very erratic habits or perform actions similar to those of intruders (such as an administrator who looks around the system and modifies system files), then an anomaly detection system will have a difficult time distinguishing an intruder from a legitimate user.

2.2.2 Misuse Detection

Misuse detection systems attempt to detect intrusions by modeling legal system behavior and flagging all actions which fall outside this model or by searching for the actual abuses of the system directly. These abuses include actions such as exploiting a bug in the operating system or taking advantage of a system misconfiguration to gain root access.

STAT (State Transition Analysis Tool) is an example of a misuse detection system which uses attack signatures to recognize intrusions. Intrusions are represented as a series of state changes from some secure state to a compromised state. The system analyzes audit data for evidence of attack transitions, issuing an alert if the compromised state is reached [8].

NSM (Network Security Monitor) is a rule-based system which analyzes network data (described in section 2.3.2) for attacks or suspicious behavior. The system has a set of keywords which it looks for in TCP packets. A rule based expert system combines the counts of all the keywords found in a particular connection to come up with an overall score [6].

Model-based intrusion detection is another idea for detecting system abuses. It is an extension to IDES (Intrusion Detection Expert System) [13] in which intrusions are described as high level models. These models are then converted into sequences of audit records which correspond to a particular attack. The attack sequences are

used to anticipate what a user would do next if they were carrying out one of the modeled attacks. Thus the system must only search for those audit records which would come next in one of the currently active attack sequences [5].

The specification-based approach finds attacks by modeling legal system behavior and flagging anything which falls outside the model. It attempts to define the legal running state of all setuid programs owned by root on a system. It can then find buffer overflows or other exploits of these programs by following the states a program enters and flagging an illegal state (one not included in the model) [11].

2.2.3 Intrusion Prevention

Intrusion prevention systems are systems which actively search a computer or network of computers for known security flaws. These systems can alert administrators about security problems before those problems are exploited by an attacker. However, as new bugs are found and new attack methods discovered, they must be updated with the information about the attacks. COPS (Computer Oracle and Password System) is one example of an intrusion prevention system. It is a collection of shell scripts which check for a variety of security flaws, including checking that files have the correct permissions and scanning the system for any files with the setuid bit set [3].

SATAN is another intrusion prevention system. It scans a network of hosts looking at the network services running on each. It then checks these services to see if they are versions with known bugs in them (such as an old version of sendmail) or if they are misconfigured (such as allowing telnet guest logins or ftp anonymous logins with access to a writable directory) [19].

2.3 Input Data Sources for Intrusion Detection

The two main sources of data analyzed by current intrusion detection systems are network sniffing data, and host-based audit data. Figure 2-1 shows a typical network with sniffer data being collected by a single dedicated machine passively listening on the network and audit data being collected on a variety of different servers and

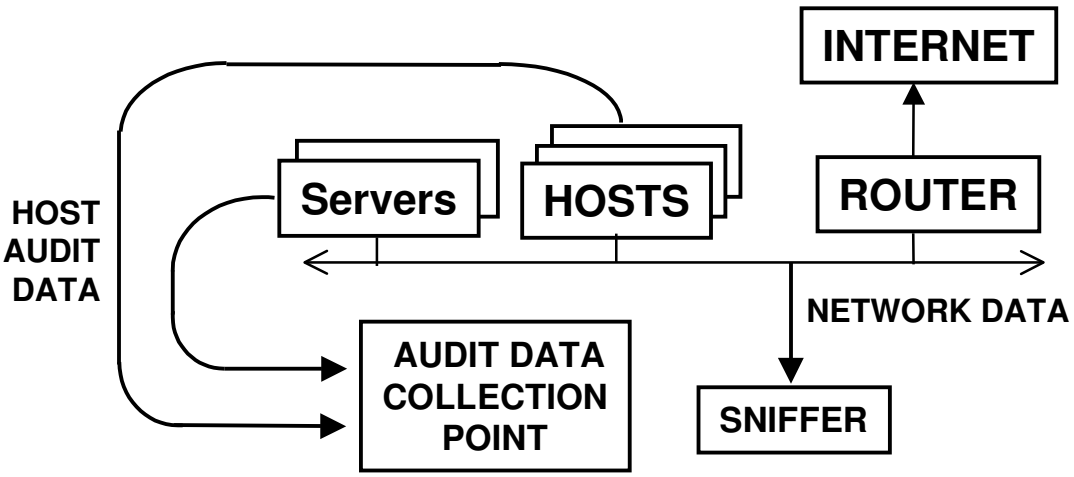

Figure 2-1: The Different Sources of Data For Intrusion Detection Systems

workstations and then exported to a central location. These different mechanisms for collection, as well as the type of information found in the different data sources, give each type of data an advantage in detection certain types of intrusions. This chapter gives a general overview of the two types of data as well as a description of the specific data used for algorithm development in this research.

2.3.1 Audit Data

2.3.1.1 Overview

Audit data refers to the log files produced by the operating system on a particular machine. These logs contain detailed information about the activities of all the processes running on a host. Generally, the logs are formatted as a list of events, called records, ranging in scale from a simple system call to a reboot or hard disk partition filling up.

Audit data provides specific details on the operation of each host, but it can be difficult to use. In a typical network environment, data will be collected on a number of machines. Logs from different machines must either be processed on those machines or exported to a central location. If the logs are processed on the machines where they are created, then the host must support the overhead of both the auditing process, and

the intrusion detection system itself. Furthermore, if a machine is compromised, an intruder could not only kill the auditing process (stopping logs from being produced), but also destroy records of previous actions. Instances of intrusion detection systems running on the hosts will either have to communicate with each other or lose the ability to correlate attacks across multiple machines.

A better approach to managing log files is to perform some prefiltering on each host and then immediately export the filtered logs to a central location for processing. This has the advantages of securing the logs, reducing computation overhead on individual machines, and allowing the data from all the logs to be merged for a more complete analysis. The log files need not be large if prefiltering is done correctly, and the primary network can be used as the transport medium without degrading network performance.

2.3.1.2 Solaris Basic Security Module

The algorithms presented in this thesis which operate on audit data all use logs produced by the Basic Security Module of the Solaris operating system. This is a kernel module which logs up to 243 different events, although the Solaris 2.5 version seems to have a bug which causes at least nine events not to be logged. These nine unlogged events are the events corresponding to the following nine system calls: fcntl(), setreuid(), setregud(), read(), getdents(), lseek(), write(), writev(), and readv().

Each event generates a record, where the record is comprised of different types of tokens depending on the event and the type of data needs to be included in the record. For example, a fork() system call (which requests that the kernel make a copy of the calling process such that two copies will be running simultaneously) has a record with a header token, an argument token, a subject token, and a return token. Table 2.2 out of the SunSHIELD Basic Security Module Guide [9] lists each type of token with a short description.

The user audit ID is a useful piece of information included in the audit records of all events which can be attributed to a specific user. It is a unique identification number assigned to every user when they login and inherited by all processes descended from

Token Name	Description
arbitrary	Data with format and type information
arg	System call argument value
attr	Vnode tokens
exec_args	Exec system call arguments
exec_env	Exec system call environment variables
exit	Program exit information
file	Audit file information
groups	Process groups information (obsolete)
header	Indicates start of record
in_attr	Internet address
ip	IP header information
ipc	System V IPC information
ipc_perm	System V IPC object tokens
iport	Internet port address
newgroups	Process groups information
opaque	Unstructured data (unspecified format)
path	Path information (path)
process	Process token information
return	Status of system call
seq	Sequence number token
socket	Socket type and addresses
socket-inet	Socket port and address
subject	Subject token information (same structure as process token)
text	ASCII string
trailer	Indicates end of record

Table 2.2: Basic Security Module Token Types

the login shell. This number allows an intrusion detection system to easily identify which user caused a specific event, even if that user has changed identities (through the su command, for example).

The log files are initially written in a binary format by the auditing process. Sun provides a tool, named `praudit`, which reads the binary log files and produce a human readable ASCII equivalent. This ASCII representation is what the algorithms working with the BSM data use. While having to convert to ASCII text slows the algorithms down somewhat, it makes coding and debugging much easier. Furthermore, once an algorithm has been prototyped and found promising, extending it to directly read the binary files would eliminate the `praudit` text conversion overhead.

2.3.2 Sniffer Data

2.3.2.1 Overview

Sniffer data refers to the information obtained from sniffing (passively listening on) a network. This approach to data collection has some definite advantages over using audit data. As shown in Figure 2-1, a single machine can gather information for an entire network (assuming it is fast enough to keep up with the network's data rate), so all the logs start off in a central location, and the overhead of collecting the data is confined to that machine. Also, since the logs are not being collected on individual machines, there is no risk of loosing information if a machine is compromised. One exception is the sniffing machine itself. If it is compromised, the logs for the entire network could be modified or destroyed. However, the sniffer can be made relatively secure since it only needs to monitor the network and not support any network services such as telnet or ftp. In fact, people have been known to cut the transmit wire in the ethernet cable attaching the sniffing machine to the network, making any network interaction with the machine virtually impossible.

Collecting data with a sniffer has one very big drawback, though. Hiding from a sniffer-based intrusion detection system can be done by simply scrambling the data before transmitting it across the network and then unscrambling it at the other end.

The scrambling could be accomplished any number of ways, such as using an encryption algorithm, a compression utility, or simply rearranging the letters using the UNIX `tr` command. The output of commands can also be scrambled and sent back to the attacker. Thus, regardless of how good the intrusion detection system is, it will have no information to work with. Furthermore, a sniffer based intrusion detection system will never be able to monitor certain services, such as ssh, which encrypt their data by default.

2.3.2.2 TCP Sessions

All network data used for algorithm development comes from TCP/IP packets [9, 18]; all other types of network traffic are ignored. Furthermore, instead of using the raw packets, the algorithms operate on ASCII transcripts, which are text representations of the data transferred between the two hosts in a TCP connection, augmented by some information from the headers of the actual packets. Finally, only data from telnet connections is used because this is the service used most frequently on UNIX system to launch real attacks. The rest of this section describes the TCP protocol and how transcripts are generated from the TCP packets.

TCP (or Transmission Control Protocol) is one of the most common Internet protocols. It provides reliable communication between two hosts over a potentially unreliable network. It is organized into connections, or sessions, where a connection can be uniquely identified by the IP addresses and ports of the machines involved. A port is simply a number associated with a process. Each host will have a process which is actively producing data it wants sent to the process on the other machine. The destination port identifies to the operating system of the machine receiving that packet the processes to which the data in the packet should be sent. The source port identifies which process the packet was sent from in case the destination processes has multiple connections open to the same machine. Figure 2-2 shows how the IP addresses and ports of the processes involved in a connection uniquely determine that connection. In this figure, a telnet client on machine A is communicating with a telnet server on port 23 on machine B. Another process on machine B, in this case a mail

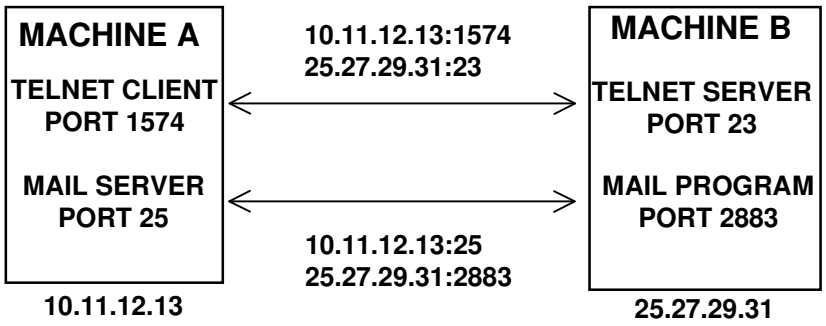

Figure 2-2: Example of Multiple Connections Between Two Hosts

program, is communicating with a mail server on port 25 on machine A. The hosts can tell which process inbound packets should be sent to by checking the port number the packet is being sent to. The source and destination IP addresses and TCP ports can distinguish a particular TCP/IP connection from all other TCP/IP connection on the network at that time.

Certain-commonly used application level protocols which use TCP as their communication medium have been assigned dedicated port numbers exclusively for use with that protocol. These protocols (such as telnet or ftp) are often called services. Thus, if one wanted to start a telnet connection to a machine, they would connect to port 23 where, by convention, that machine's telnet server would be listening (as the telnet server was in figure 2-2).

2.3.2.3 Transcript Creation

As mentioned previously, the sniffer data available is in the form of text transcripts, which are ASCII representations of the data in the TCP packets. These transcripts were collected by a baseline system running on about 50 computer networks over a period of about 13 weeks. This section gives a brief description of how the baseline system works and where the transcripts came from. See section 2.4 for a more detailed

```
}{{{ {!{"{'}{#zPp|$zXTERMpz#plato:0pz'DISPLAYplato:0p$}|mary
mary12
ls -a
pwd
exit
```

Figure 2-3: Init File For a Transcript Generated by the Baseline System

explanation of the baseline system.

Each transcript corresponds to one TCP connection, and is made up of two files. The init file contains all the information the initiating host (the client) transmitted over the course of the connection. For a telnet connection, this corresponds to everything a user types. The dest file contains all the data that was transmitted by the destination host (the server). This is the data displayed on a user's screen. Any non-printable characters (ASCII characters not between 32 and 126) are ignored, except for the escape character (ASCII 27) which is represented by "^[". The header of the transcript gives the source and destination ports and IP addresses, the names of the machines (or "unknown" if they could not be ascertained), the times the connection started and ended, and the keywords found by the baseline system. Figure 2-3 shows the body of the init file of a sample transcript from a session in which the user `mary` logs in with the password `mary12` , executes an `ls` command and a `pwd` command in her home directory, and then logs out. Figure 2-4 shows the body of the dest file. The strange sequence of characters on the first line in each file and the braces before the login prompt and the tilde after it in the dest file are what is left of the negotiations between the telnet client and server when the connection was set up after the non-printable characters have been stripped out.

Figure 2-7 shows how transcripts are generated. The sniffer module of the baseline system captures all network traffic and then filters it, only keeping data for the services the system is configured to monitor. Every day the expert system module processes all data collected. This system assigns a warning level from zero to ten to each connection. Transcripts are then generated for a certain percentage of the high scoring connections and sent to human analysts to check for attacks. Transcripts are

```
}}}#}'}${~ ~!~"|~$zpz#pz'p

UNIX(r) System V Release 4.0 (pascal)



{}login: ~mary
Password:
Last login: Mon May 25 00:49:02 from plato.eyrie.af.m
Sun Microsystems Inc.   SunOS 5.5.1     Generic May 1996
mary@pascal: ls -a
./       ../      .cshrc
mary@pascal: pwd
/.sim/home/mary
mary@pascal: exit
mary@pascal: logout
```

Figure 2-4: Dest File For a Transcript Generated by the Baseline System

also archived for a certain period of time. The data used for testing and developing algorithms in this thesis consists of all high scoring transcripts in a 13 week period. Figure 2-5 gives a breakdown of types of telnet transcripts seen during this time. This figure shows that transcripts containing nothing but failed login attempts make up the biggest section of the data at 35%.

2.3.2.4 Attacks in Data

This data is valuable because it contains a large volume of normal user activity as well as real attacks. Normal activity is important for both training algorithms and analyzing their false alarm rates. Real attacks are useful both for the actual break-in techniques used as well as all the actions the intruders take once they have compromised a system. Since some intrusion detection systems look not only for the attacks themselves, but also for the kinds of actions intruders normally perform, it is important to have real examples of these actions for testing.

A total of seventeen apparently illegal transitions to root were found in this data by other researchers at MIT Lincoln Laboratory using a combination of automatic

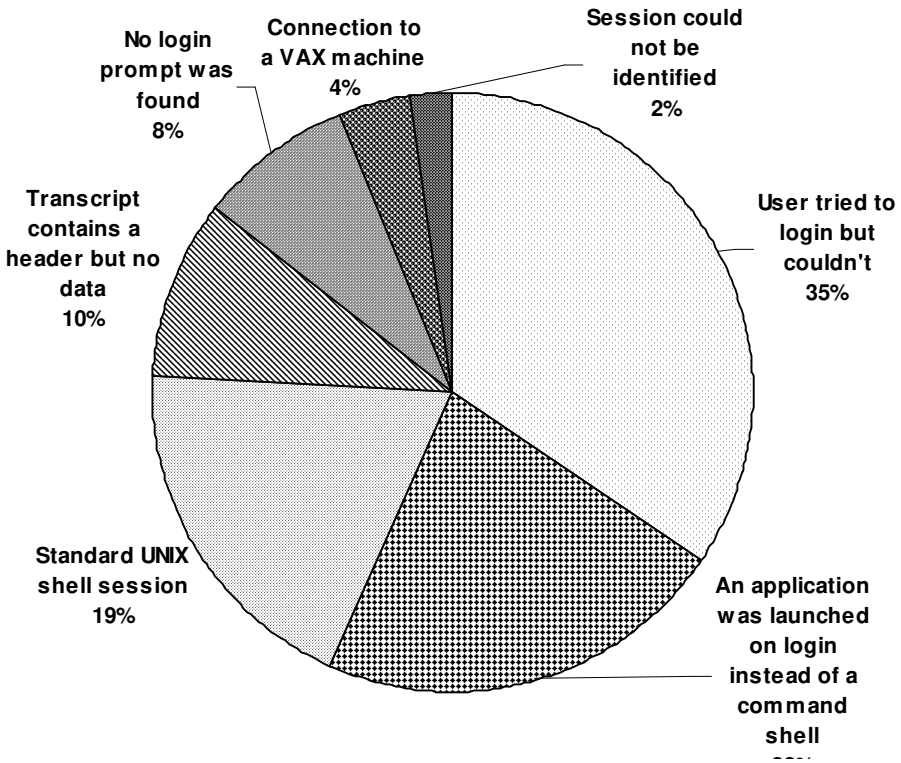

Figure 2-5: Breakdown of Telnet Transcripts

search and hand analysis [20]. Five of these transitions were actual attacks while the remaining 12 appear to be instances of system administrators using a setuid root shell to get root permissions without entering a password. While these setuid root shell transitions are technically illegal, they were not used as attacks when training or testing the algorithms because the actions performed after obtaining root privileges are legal system administration activities. In addition to the five attacker transitions to root, two other instances of malicious activity were found by prior analysis of this data. These seven attacks found in the sniffer data (six different types in all) are the attacks used to train and test the algorithms presented in this thesis. The following is a brief description of each attack:

loadmodule A user exploits a bug in the `loadmodule` binary related to the internal file separator (IFS) variable to gain root access.

PERL(2) This attack was actually used twice on two different hosts. It consists of an attacker exploiting a bug in the PERL interpreter to set the user ID of a shell to root, thereby giving the attacker access to all files on the system.

sniffer A user logs in and becomes root legally by using the `su` command and entering the root password. However, once he/she becomes root, the attacker proceeds to check the log files produced by a packet sniffer apparently running covertly on the system.

custom system library This attack actually has two parts. Exploiting the fact that the system has an anonymous FTP server with a writable directory, an attacker uploads a corrupted version of the libr.so system library to this directory. Next he/she telnets to the machine with an option telling the telnet server to link to the corrupted version of the library. This version causes the telnet server to bypass the login and password prompt and immediately give the attacker a root shell.

setuid root ksh In this session, the attacker doesn't actually exploit a vulnerability of the system, but instead becomes root through a back door apparently left

from a previous connection. Hidden in the `.netscape` directory is a copy of the `ksh` shell but owned by root and with the setuid bit set. Executing this binary launches a new k-shell running with root permissions.

HP remsh This attack uses `remsh` to exploit a bug in usr/etc/vhe/vhe_u_mnt on an HP computer. On completion, it gives the attacker a root command shell.

2.3.2.5 Problems With Transcripts

Two problems inherent in the data made analysis difficult. The first is that this data is not a valid cross section of all network traffic. Only transcripts which were scored highly by the baseline system were available, corresponding to roughly 1 out of every 100 telnet connections. Without access to the remaining 99% of the data, it is impossible to determine how many attacks were missed by the baseline system and how many of those missed attacks would be found by other algorithms.

A second problem has to do with an apparent bug in the code used by the baseline system to generate transcripts. Specifically, transcripts often contain characters or groups of characters repeated multiple times. Often a word in both the init and the dest file (such as a command typed on the client and echoed back by the server) which should be the same will have characters doubled in one file but not the other. Figure 2-6 shows an example of some of the different ways the character doubling can manifest itself. The first column shows the lines corresponding to a login if no characters were doubled. In the second column, individual and groups of letters are doubled in the login name in the dest file. This kind of doubling makes it difficult for algorithms to match words in the two files. The third column shows letters in the password from the init file doubled. Detecting doubling in the password is very difficult because, by definition, a password is an arbitrary arrangement of characters. Finally, the fourth column shows a transcript in which the entire username in the init file, including the newline character at the end of the line, has been doubled. If an algorithm were looking for the password on the line directly below the username, it would mistake the username for the password.

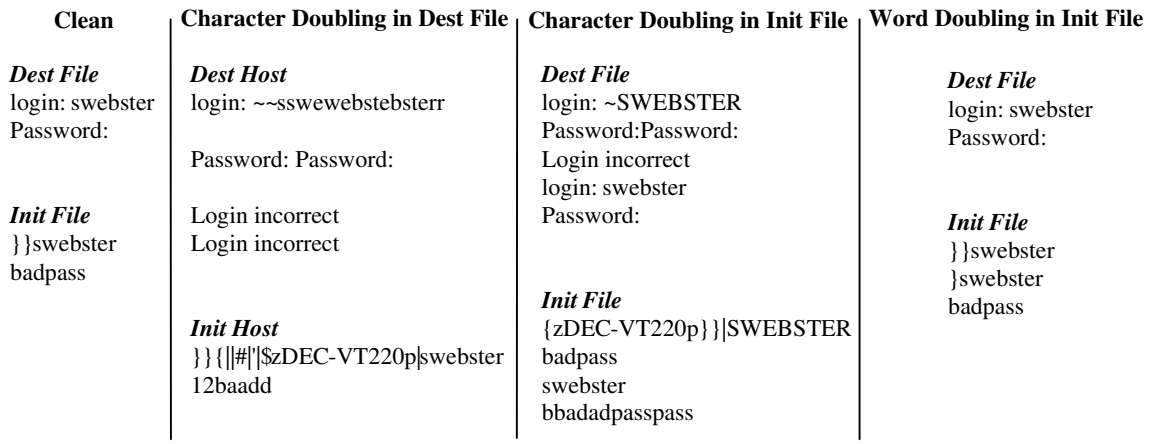

Figure 2-6: Examples of Character Doubling in Transcripts

One possible explanation for the doubling is that retransmitted packets are not discarded when the transcripts are created. In a few cases, characters are reversed, indicating that packet reordering is not performed either. Other times, however, every character in one file appears to be doubled. This phenomenon can not be explained well by the theory that retransmitted packets are not dropped. Whatever the reason, character doubling adds a lot of complexity and detracts from algorithm performance. Fuzzy string matches must be performed, and often a word is so corrupted in one file that a fuzzy match fails. Appendix A discusses a transcript generation utility developed as part of this research named tcptrans. This utility produces transcripts free from any character doubling.

2.4 The Baseline System

Figure 2-7 shows how the baseline system collects and processes data. This system, used as a standard for performance, is a sniffer-based expert system which scores connections based on keyword counts. It is derived from NSM[6] and has three main phases of operation: sniffing, packet sorting, and scoring. In the first phase, a sniffer collects raw packet data. Typically, the sniffer is run continuously, with a brief break to score the data every day. While running, the sniffer filters the network traffic writing all packets which pass the filter to temporary files. The filter is determined

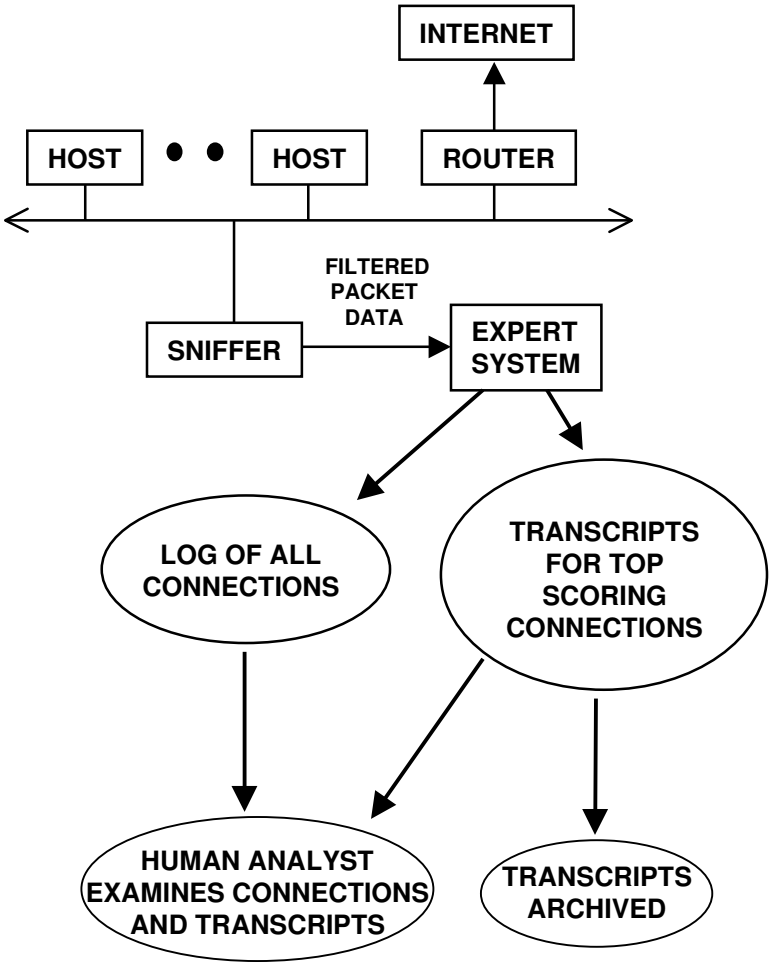

Figure 2-7: Normal Operation of the Baseline System

by a configuration file that specifies which TCP services the system should monitor. All TCP packets not corresponding to one of these services and all non-TCP packets are ignored.

When an administrator wants to analyze the collected data, he/she starts the sorting and scoring program. The sorting program arranges packets in the temporary files into their respective connections. The data from each connection is then searched for instances of certain keywords. These keywords (40 in all) are used by an expert system to determine its component of the total score. Some of the keywords correspond to known attacks, such as system binaries which are rarely used

by legitimate users, but older versions of which have known bugs ("loadmodule" for example). Other keywords, such as "daemon:", are aimed at catching attackers as they explore the system. This keyword corresponds to an entry usually found in the `/etc./passwd` file and would show up if the attacker viewed that file on his or her screen.

The total score for a connection is obtained by combining the scores from three different components. These are the expert system, the profile component, and the attack model component. Each of these components produces a score from 0 to 10. The scores are then combined using the following formula:

$$total\ score = (1/4 \times profile) + (1/8 \times model) + (5/8 \times expert)$$

where $profile$ is the profile score, $model$ is the attack model score, and $expert$ is the expert system score.

The expert system is a rule-based system which takes as input keyword counts and produces a score indicating how dangerous the corresponding connection is. For some services, the expert system will use other information in addition to the keyword counts when assigning a score. One example is that a finger connection will receive a high score if the finger query is over 500 bytes long, regardless of the content (this rule is designed to detect attackers attempting to overflow a buffer in the finger daemon).

The profile component looks at the IP addresses of the two machines in the current connection and the service they are using and assigns a score based on how often these two computers have communicated in the past seven days. Specifically, the score starts at 10 and, depending on how many days ago a connection was made, a fixed amount is subtracted from the score. Table 2.3 gives the weights which are subtracted from the score for a connection on that day. This table shows, for example, that if a connection was made using a given service yesterday, the profile score for a connection between the same two machines on the same service today will be 1.5 less than if yesterday's connection had not been made. Thus, a connection from a computer not seen before is considered more dangerous than one from a well known

Day	Weight
0	2
-1	1.5
-2	1.25
-3	1.125
-4	1.125
-5	1
-6	1
-7	1

Table 2.3: Day/Weight Pairs

machine.

Finally, the attack model component assigns a value to this attack based on the hosts and service involved. The first part of this score is based on the level of trust for each machine. A file of known hosts lists host IP addresses and a security rating between one and ten, where a ten indicates that host is very highly trusted. An untrusted host gets a rating of zero. If the initiating host has a higher rating than the destination host then this part of attack model score is zero. Otherwise it is 4/9 times the difference between the destination host's and the initiating host's security ratings. The second part of the score is based on the threat capability of the service used. The third and final part of the score is based on the authentication strength of the service. These last two values both range between one and ten and are static values stored in a file of known services. Telnet, for example, has a threat capability score of 10 and an authentication score of 7. The equation for calculating the overall attack model score is:

$$attack\ model\ score = (4/9 \times security) + (3/9 \times threat) + (2/9 \times (10 - auth))$$

where *security* is the host security score, *threat* is the service threat capability score, and *auth* is the service authentication strength score.

2.5 ROC Curves and Purity

An ROC (receiver operating characteristic) curve is a graph which shows how the false alarm and detection rates of a system vary. The graph has the detection rate (as a percentage) on the vertical axis and the false alarm rate (as a percentage) on the horizontal axis. To generate an ROC curve for an intrusion detection system, the system must first assign scores to a set of transcripts containing both attacks and normal behavior. These scores are then divided into two lists such that all the attacks are in one list and all the normal connections in the other. The lists are then sorted with the higher scores first (assuming that the intrusion detection system tries to give attacks high values and normal connection low values). The curve is constructed by selecting different threshold, or cutoff, values. For a given threshold, all transcripts with a score above that value are the transcripts the intrusion detection system labels as attacks. Thus, given a threshold, the number of detections (attacks with a score above the threshold) and the number of false alarms (normal connections with a score above the threshold) can both be determined. These two values are converted into percentages and plotted as one point on the ROC. After the points for all possible threshold values have been plotted, they are connected to form the ROC curve itself. Table 2.4 shows some sample data with the scores sorted into the two lists "attack" and "normal". Figure 2-8 shows a sample ROC. The labels at each point in the form of "#<t<#" show the threshold values which correspond to that point.

A system which randomly scores all connections should have an ROC with a diagonal line from 0% detection and 0% false alarm rate in the lower left corner to 100% detection and 100% false alarm rate in the upper right corner. A good system which gives most attacks a high score and most normal connections a low score have an ROC which climbs very sharply to a point with a high detection rate and a low false alarm rate and then levels out. One metric derived from an ROC plot is the area under the curve, where 100% area is a perfect system and 50% is random guessing.

Since the detection and false alarm rates are given as percentages, the actual number of detections and false alarms are hidden. The metric of purity is used to

Attacks	Normal
9.8	7.3
9.8	7.0
8.3	7.0
6.2	5.4
4.2	4.7
	4.7
	3.0
	3.0
	2.1
	2.1

Table 2.4: Sample ROC Data

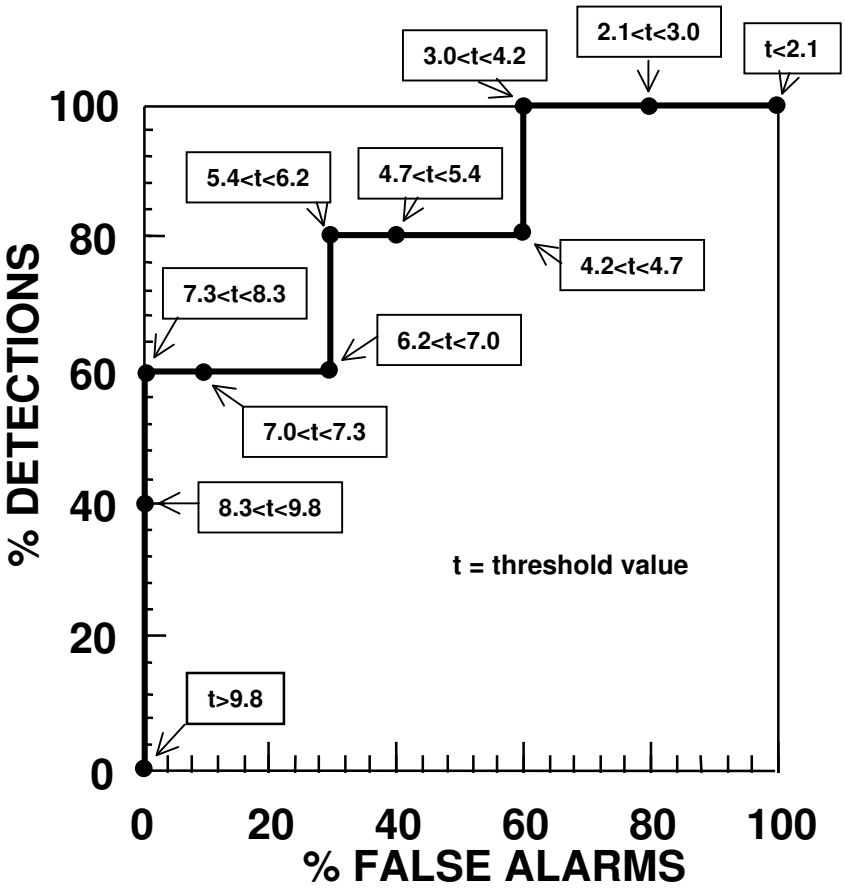

Figure 2-8: Sample ROC Plot

give an indication of these numbers. It is computed as the ratio of the actual number of attacks detected to the number of false alarms for a given threshold value. Typically the threshold is chosen to be the value which gives a certain detection rate (such as 80% detection). If a human analyst has to look over all high scoring connections to separate the attacks from the false alarms, the purity gives an indication of the workload associated with finding each attack. For example, a system may have a 10% false alarm rate at 100% detection. However, if that system sees 100,000 connection a week and only ten of them are attacks, then the purity is only 1/1000. This means that every week an analyst will have to look through 1000 transcripts to find each attack. One goal of the work presented in this thesis is to develop systems which increase the purity of the baseline system to the 1/10 to 1/100 range for large amounts of data and realistic attacks.

Chapter 3

Password Guessing Detector

In looking at the different kinds of telnet connections which were scored highly by the baseline system over 13 weeks, a surprising number of them are sessions without a valid login (see figure 2-5). Upon further investigation, it was found that the string "Password incorrect" is weighted very highly by the baseline system (expert system score of 8.5 for one occurrence) in the absence of the string "Last login", which would indicate a valid login. Even with a valid login, one occurrence of "Password incorrect" will still score 6.5. This behavior is an attempt to sensitize the baseline system to password guessing attacks, either at login or elsewhere in the session (the 'SU' prompt for example). However, in looking over a few transcripts without valid logins, all of them seemed to be the result of legitimate users forgetting their passwords. Thus, the question was raised of whether or not there were actually any large scale dictionary or doorknob rattling attacks in the current data to justify the high weighting of the "Password incorrect" string.

A new algorithm, called the Password Guessing Detector, was developed to identify dictionary and doorknob rattling attacks across multiple connections with a low false alarm rate. The algorithm was developed both for its own sake, and to use in analyzing the data at hand to see if these attacks actually occur in the real world. One side effect of the "Password incorrect" string being weighted so heavily is that the data is almost guaranteed to contain all connections in which a password was mistyped for the 13 week period, making this analysis reasonably complete. This

chapter gives an overview of doorknob rattling and dictionary attacks, describes the algorithm and how it tries to spot these attacks, and then presents the results found when the algorithm was used to analyze the 13 weeks of data.

3.1 Dictionary and Doorknob Rattling Attacks

Dictionary and doorknob rattling attacks are both attacks in which an intruder is trying to gain access to the system by finding a valid username and password. This type of attack can be applied to any network service which requires a username and password, such as telnet, rlogin, ftp, or pop mail servers. Dictionary attacks are large-scale attacks in which hundreds or thousands of passwords are tried with one or more usernames in an attempt to exploit the fact that people will often choose simple, easy to remember passwords. Thus, instead of trying all valid passwords (about 6×10^{15} assuming that passwords can contain any of the 94 printable ASCII characters and are up to eight characters long), the attack can focus on common words, such as "wizard" or "pyramid", or easily typed character sequences, such as "asdfjkl;". Considering that, even at three attempts a second, trying all passwords would take about 64 million years, dictionary attacks have no choice but to limit themselves to guessing simple passwords. However, given no a priori knowledge about a password except that it is a common word, the attack still must try on the order of thousands of passwords before it is likely to find the correct one. Thus, these attacks would be carried out by automated scripts which repeatedly log into a machine and try username/password pairs until a match is found. A variation on this attack is to download the password file to a local machine and then run a password cracking program on it, such as CRACK [14]. These programs guess passwords by encrypting trial words and seeing if the encrypted test strings match any encrypted passwords from the password file. This approach has the advantage of being much faster (orders of magnitude) as well as not leaving failed login messages in system logs. However, the password file must be obtained before this variation of the attack can be carried out.

Doorknob rattling attacks are attacks which poke a system in the hope of find-

open	help	games
guest	demo	maint
mail	finger	uucp
bin	toor	system
who	ingres	lp
nuucp	visitor	
manager	telnet	

Table 3.1: Names of Commonly Attacked Accounts

ing accounts present by default that have not been removed or accounts which are misconfigured. Examples of this include looking for a "guest" account which will allow anyone to log on, but with reduced permissions, or a "root" account with no password. Table 3.1 lists some common account names which attackers often try [4]. Once an attacker has gained access to the system, even with reduced permissions, the security of the system is seriously compromised.

The difference between doorknob rattling and dictionary attacks is that doorknob rattling attacks require many fewer connections to a single host. Since there are only about 20 common account names for the attackers to try, it is possible for the attack to be carried out by hand if only a few hosts are to be checked. The tradeoff is that, while it is quite likely that at least a few passwords on any given system will be simple, there is no guarantee that any default or misconfigured accounts will exist. An attacker may have to try tens or hundreds of different hosts before a doorknob rattling attack is successful.

3.2 The Password Guessing Detector

This algorithm looks at usernames and passwords typed at the login prompt and attempts to differentiate between valid users who have forgotten or mistyped their password and attackers trying to guess the password for an account. The algorithm could be extended to also look for password guessing at the `su` prompt or in other TCP services besides telnet.

The Password Guessing Detector operates by making a few assumptions about the nature of dictionary and doorknob rattling attacks. First of all, it assumes the probability of a dictionary attack succeeding on its first attempt is almost zero. The algorithm thus ignores connections which contain a successful login, even though the user may have entered a few incorrect passwords or usernames. This eliminates all the sessions in which a legitimate user mistypes a password once or twice. Even though the actual login of a successful attack may be lost, all the other connections in which the attack fails to find a valid password will still be considered.

The algorithm also assumes that the passwords being guessed will be simple passwords (as defined below). As discussed in the previous section, an exhaustive search of all possible passwords is impossible due to the size of the search space.

The final assumption is that an automated script would not try the same password more than once. When a legitimate user tries to log in with the wrong password, they generally try the password a few times to make sure that they are not mistyping it. An automated script will not make these kinds of mistakes (unless intentionally programmed to do so).

A separate algorithm is used to check the integrity of the password (see section 6.2 for a full description). This program takes as its inputs a password and a username, and outputs a value from zero to six indicating how difficult the password would be to guess. A value of 3 or less indicates that the password matched or was very similar to either the username or a dictionary word.

Based on the above assumptions and the password scoring algorithm, the password guessing detector selects connections to consider for possible attacks as follows:

- Only telnet connections in which a successful login never occurred were considered.
- Connections in which the same username and all strong passwords are tried are ignored.
- Connections in which the same username and password is tried two or more times are ignored, regardless of the strength of the password.

The connections which meet the above criteria are then summarized and printed to a file. The file is formatted such that the connections are sorted first on the number of connections originating at each source IP address. For each initiating machine the connections are then sorted on the number of connections to each destination IP address. Finally, for each destination IP address, the usernames and passwords tried are listed. Because of the number of connections required in launching a dictionary attack, any such attacks should show up near the top of the list, even if the attacker ran the attack from a number of machines. Also, while all of the connections meeting the above criteria were included in the file, it would be a trivial extension to implement a cutoff such that connections from a host with fewer total connections than the cutoff value would be ignored.

3.3 Password Guessing Detector Results

After running all four months of data through the above algorithm (about 23,000 telnet connections without a successful login) and examining the results, no large scale dictionary attacks was found. To be sure the algorithm was not accidently filtering out attacks, one pass through the data included all connections without valid logins regardless of whether or not strong passwords or the same password were being tried. Some incidents of people trying to log in to generic accounts ("guest", "admin", "ftp", etc.) were found, but even those were relatively rare and none of them appeared to be part of a large attack spanning multiple hosts. Table 3.2 shows the total number of connections and the number of those that appeared malicious by hand examination. The "inside" label correspond to machines on the networks the baseline system was setup to monitor while "other" represents all other machines (those not on the monitored networks and those whose names could not be resolved when the transcripts were created and were thus listed as unknown). Judging from this data, dictionary attacks are relatively rare, if not non-existent. The only doorknob rattling attacks appear to be casual ones against one or two hosts at the most. Figure 3-1 shows an example of a series of three connections that appear to be part of a doorknob

		Destination: inside	Destination: other
Source	inside	6,742 Total 0 Malicious	3.966 Total 11 Malicious
	other	9,042 Total 23 Malicious	3,205 Total 0 Malicious

Table 3.2: Password Guessing Algorithm Results

rattling attack. The attacker appears to be going down a list of account names and using the next account name as the password for the current account. While it is not clear this strategy is actually effective, this activity is still very suspicious in light of the usernames being tried.

```
204.134.203.82 (?.mtn.micron.net) (source)
--------------
        USERNAME        PASSWORD
        --------        --------
(destination)   07/24/1996 16:40:48 -- x.x.x.x (same.target.host)
        mountfs         adm
        adm             uucp
        uucp            anon
(destination)   07/24/1996 16:42:50 -- x.x.x.x (same.target.host)
        user            demo
        demo            admin
        admin           sync
        sync            guest
        guest
(destination)   07/24/1996 21:52:53 -- x.x.x.x (same.target.host)
        turkey          dalick
        ruth            fcc
        fcc             uucp
        uucp
```

Figure 3-1: Sample Output of the Password Guessing Detector for a Series of Suspicious Connections

Chapter 4

Keyword Reweighting System

As mentioned above, the string "Password incorrect" is weighted very heavily in the baseline system in an attempt to catch dictionary and doorknob rattling attacks. In analyzing the 13 weeks of data with the Password Guessing Detector, however, no evidence of any dictionary attacks was found. A few doorknob rattling attacks were found, but they accounted for a very small percentage of the total connections. Thus, the 36% of the telnet transcripts which do not contain a valid login are almost exclusively false alarms. This suggests that the current weighting of the keywords by the baseline system is not optimal given the real-world normal behavior and attacks. This chapter examines the performance improvement that can be gained by using a neural network trained on both normal data and attacks to score the connections. The first section describes the neural network and how it is trained and tested. The second section gives the results from using the neural network and compares its performance to that of the baseline system.

4.1 The Neural Network

The neural network used to score the connections is a single layer perceptron with 40 input nodes (one for each keyword used by the baseline system) and two output nodes (the first representing the probability that this session is normal and second representing the probability that it is an attack). The keyword counts for each transcript

are combined to form an input vector and these vectors are fed to the neural network. The network was trained and tested on 20,000 normal transcripts and seven attack transcripts (the attacks used are those listed in section 2.3.2). Keyword counts were taken from the transcript header to ensure that the both the neural network system and the baseline system used the same counts for each transcript. The neural network was trained and tested using a classifier training software package called LNKnet[12]. To compensate for the limited number of attack transcripts, the data was broken up into seven groups, or folds. Each fold contained 1/7 of the normal sessions and one of the malicious sessions. During each evaluation, six of the seven folds were used for training and the last fold for testing. This process was repeated seven times with the network being retrained each time, so that each fold was used for testing one time and for training six times. This scheme (called seven-fold cross validation) made it possible to gather test statistics on all the data without ever training and then testing the network with the same data. Figure 4-1 shows a picture of part of the neural network after training. The lines from the input to the output nodes represent the effect that the input nodes have on the output nodes. Solid lines represent positive weights, or reinforcement, and dashed lines represent negative weights, or inhibition. The thicker the line, the greater the magnitude of the weight it represents.

4.2 Results

Figure 4.2 shows the ROC's and purity for both the neural network system and the baseline system. As was suspected, the weights assigned to the keywords in the baseline system are not optimal, and reweighting the keywords based on real data provides a significant reduction in false alarm rate and almost an order of magnitude improvement in the purity. The area under the ROC curve for the neural network increased 21% over that of the baseline system. The purity at 90% detection improved from 1/2300 to 1/260. These statistics show that the neural network system is able to detect all the attacks at a much lower false alarm rate than the baseline system.

The large improvement in performance provided by the neural network system

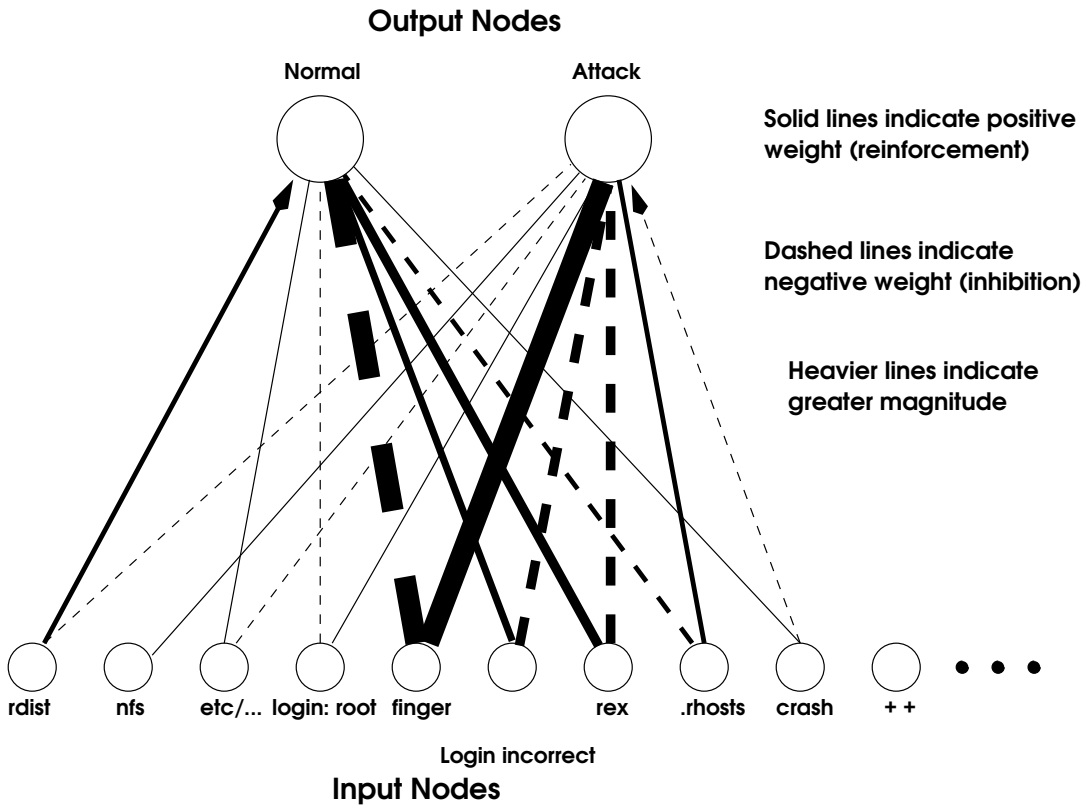

Figure 4-1: Neural Network Used to Score Connection

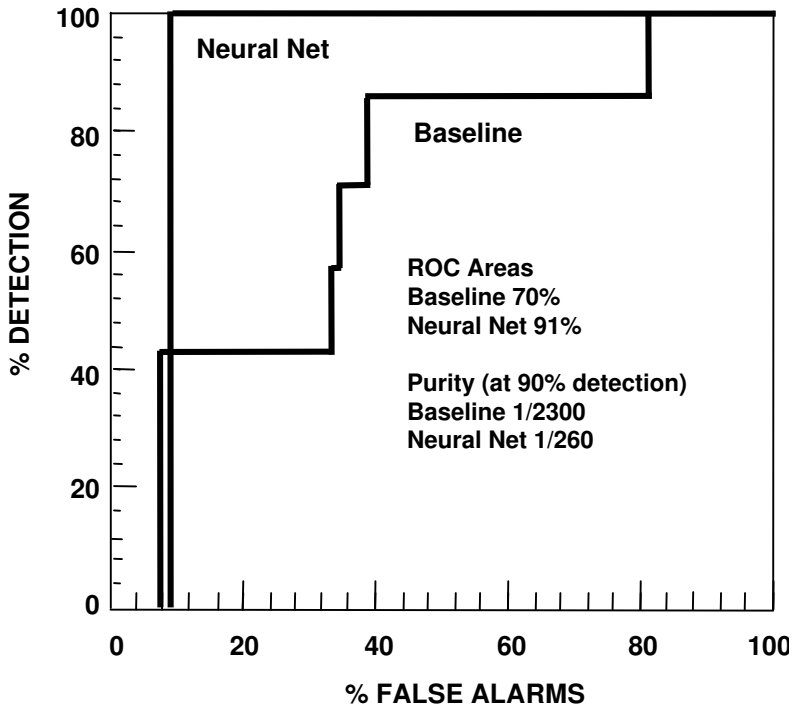

Figure 4-2: Performance of Baseline System and Neural Network

is obtained through modeling both normal and malicious behavior instead of simply modeling malicious behavior as in the baseline system. A keyword associated with an attack may have a large positive weight connecting it to the "attack" output node while a keyword indicative of a normal session would reinforce the "normal" output node. This weighting can be seen in figure 4-1. For example, the keyword "finger" is highly correlated with attacks, while the keyword "rdist" was found mainly in normal sessions. The keyword "Login incorrect" actually weakly reinforces the "normal" score and weakly inhibits the "attack" score.

Chapter 5

Bottleneck Verification Prototype

Bottleneck verification is an approach which allows a system to detect attacks without any a priori knowledge about the attack. It involves monitoring both the current state of users or processes as well as the legal transition paths to other states. This chapter gives an overview of the bottleneck verification concept and then describes a prototype system that implements this approach by using audit data to monitor the command shells launched by users and detect illegal user to root transitions.

5.1 Bottleneck Verification Overview

The bottleneck verification approach applies to situations in which there are only a few, well defined ways to transition between two groups of states. Figure 5.1 shows such a system with states represented by points and transitions represented by lines. In this hypothetical system, there are many highly connected user states and many highly connected root states. However, there are very few connections between these two groups of states. If a system can ascertain the state of a particular user, as well as monitor all legal transition paths from that state to other states of interest, then the system can determine when that user has illegally transitioned to one of those other states by noticing the change to the new state without the use of any of the corresponding legal transition paths. This approach allows the system to detect attacks which result in a change in user state without knowing anything about the

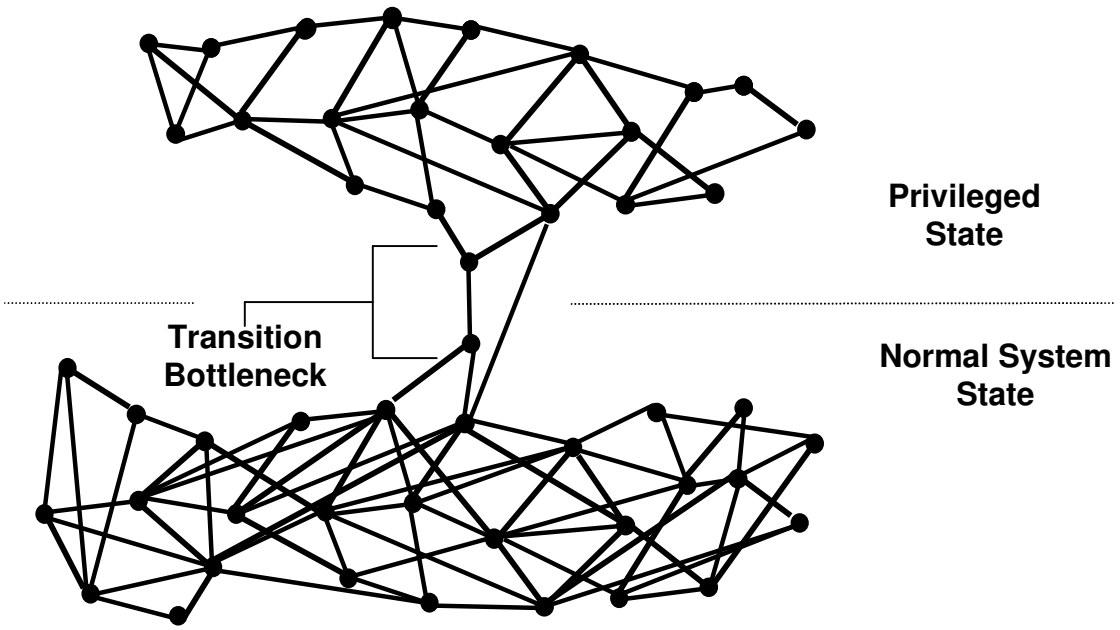

Figure 5-1: Example of a Bottleneck Between Two Groups of States

attack. In effect, the system notices the absence of legal behavior instead of the presence of illegal actions.

5.2 The Shell Tracking Algorithm

The Shell Tracking algorithm is an implementation of the bottleneck verification approach using audit data from Sun's BSM to monitor the command shells launched by users on a system. It monitors the user and root, or superuser, states. If an individual is in the user state, he/she can typically only get a command shell with root privileges by using the su command and entering the root password. Thus, if the system can detect when a shell is being launched, determine the permissions of that shell, and detect all successful uses of the su command to get to root, then illegal transitions to root can be detected by looking for root shells launched by users who should not be in the root state. Root shells were chosen as the metric for detecting a state change because they are relatively easy to monitor and because, historically, the majority of attack scripts distributed over the Internet launch a root shell on

completion. This is changing, however, and newer scripts have started adding a user to the password file instead. Detecting these new attacks is covered in section 8.3 under future work.

The system specifically looks for a root shell being launched from a process where the owner of the process did not either log in directly as root (and thus start with root permissions) or execute a valid "SU" (valid means that the user correctly entered the root password) command anywhere in the process' ancestry. It detects a shell being launched by monitoring all fork() and exec() system calls. If the name of the process which is being started in the exec() call is one of tcsh, sh, bash, csh, or ksh, the system flags this process as being a command shell. Given the specific type of shell and the real and effective user id of the shell's process, the algorithm can deduce whether or not it is a root shell. For example, some shells, such as "sh", determine their permissions from the real user id while others, such as "ksh", look at the effective user id.

The system keeps a table of all currently running processes where each entry contains a process id and a root bit, which indicates whether or not that process is allowed to launch root shells. The exit() and kill() system calls are monitored so that processes can be removed from the table when they are terminated. When a new process is created and added to the table, it inherits the root bit value of its parent. A valid su command will set the root bit of that process to 1, indicating that future processes may launch root shells. An illegal root shell will also set this bit to 1, so that only one alarm will be reported per intrusion.

The system was tested using a buffer overflow and suid shell script attack, both of which it found. The buffer overflow was in the `eject` binary. The attack script overflowed the buffer with code to launch a shell, which has root permissions since the `eject` binary runs as root. The setuid shell was a copy of `ksh` owned as root and with the setuid bit set. When launched, it runs with root privileges. The false alarm rate has not been extensively tested (although it is believe to be low), because of the lack of a large volume of normal audit data. The system has a few advantages over the baseline system in detecting intrusions. The first and most significant is that it

can detect a wide range of attacks without any prior knowledge about those attacks. Generally speaking, if the base line system does not have a keyword associated with a particular attack, it relies on detecting an intruder's suspicious actions. The keywords associated with these suspicious actions tend to introduce false alarms, since system administrators and attackers often both make changes to the same system components (the password file, configuration files, etc.).

A second advantage over the baseline system is that the Shell Tracker runs in real time. As the audit logs are created, a PERL script implementing the shell tracker continuously monitors the log files and pipes new records through `praudit` as they are written. Furthermore, because only a few audit events are used by the algorithm (fork(), exec(), exit(), kill(), etc.) the amount of data produced by the auditing system is relatively small. This allows the Shell Tracker to keep up with system activity, even under a relatively heavy load.

Finally, the Shell Tracker gains an advantage simply through its use of audit data instead of sniffer data. This advantage is that efforts to hide from network sniffers will not affect the Shell Tracker as the attackers actions are still recorded in the audit data. It is, however, possible for an attacker to hide from the shell tracker by either having the attack code perform some action other than launch a shell or by renaming the shell. Both of these issues are addressed in section 8.3 as suggestions for future work.

Chapter 6

Intrusion Prevention Algorithms

Intrusion prevention algorithms, instead of attempting to detect actual attacks, are concerned with finding poor security practices which may make a system more difficult to defend. This chapter presents two intrusion prevention algorithms, both of which analyze sniffer data. The first is the Warning Banner Finder, which checks for the presence of a valid warning banner. The second is the Password Checker, which analyzes the passwords of users as they log in to detect accounts with weak passwords.

6.1 Warning Banner Finder

6.1.1 Description of a Valid Warning Banner

The main purpose of a warning banner is to protect system administrators from invasion of privacy lawsuits by informing users that their actions on a particular system are being monitored and to facilitate prosecution of attackers if they are caught breaking into systems. Based on these ideas and on a sample warning banner provided by the Department of Justice, the following three conditions were chosen as the definition of an acceptable warning banner:

- The warning banner must state that all users of this system are subject to monitoring.

- The warning banner must further state that use of this system implies consent to such monitoring.
- Finally, the warning banner must make some statement about who is allowed to use this system.

These criteria are meant to insure that anyone breaking into the system can not claim either that their privacy was invaded by the monitoring or that they did not know it was a private system and that they were not supposed to be on it. The following are two examples of warning banners which meet the above criteria:

```
SYSTEM FOR USE OF FULL-TIME AJAX CORP. EMPLOYEES ONLY. DO
NOT DICUSS, ENTER, TRANSFER, PROCESS, OR TRANSMIT COMPANY
CLASSIFIED INFORMATION. USE OF THIS SYSTEM CONSTITUTES
CONSENT TO SECURITY TESTING AND MONITORING. UNAUTHORIZED USE
COULD RESULT IN CRIMINAL PROSECUTION.
```

```
Unauthorized use is a CRIMINAL OFFENSE.
Use constitutes consent to Security Testing and Monitoring
```

6.1.2 The Warning Banner Finder Algorithm

This algorithm uses a binary decision tree and keyword counts of five words and phrases to determine if a warning banner exists in a transcript. The decision tree was constructed on the assumption that the keywords found in warning banners would be very rarely found in the first 100 lines of a transcript that did not contain any sort of warning banner. Thus, a system which used keywords found only in the banners should easily be able to distinguish between a transcript with a valid warning banner and one with no warning banner at all, given that the transcript with no warning banner would have zeros for most of the keyword counts. The hard problem, then, is differentiating between acceptable and unacceptable warning banners.

Based on this assumption, a number of transcripts were gone through by hand and the warning banners were extracted and placed in a file. All phrases from the text of the warning banners with between one and five words in them (i.e. n-grams of length one through five) were constructed from this file. The 500 most frequently occurring phrases were then used as keywords, and a set of keyword counts for each of 200 training transcripts was generated. These sets were turned into feature vectors which could be processed by LNKnet [12] and used as inputs to the various classifiers it implements.

In trying a few different classifiers, it was found that a binary decision tree[12] performed as well as any of the others, and lended itself to easy implementation in PERL. Based on this, automatic feature selection was performed (also using LNKnet) with a binary tree as the selection algorithm. It turned out that the top five keywords were the only ones needed to classify the training data with 100% accuracy. The decision tree that LNKnet produced is shown in figure 6-1 with the keywords and threshold values. All the threshold values turned out to be .5, indicating that simply the existence of a keyword would cause the algorithm to take the corresponding right branch. For example, a transcript which contains the phrases "monitoring", "only", and "information" in the first 100 lines would take the first three right branches and be classified as having a valid warning banner.

The final algorithm is a PERL script which takes a transcript as input, searches through the first 100 lines counting instances of the keywords and then feeds these counts into the decision tree. The decision tree then returns a binary answer indicating whether or not the transcript contains a valid warning banner. This script was run on 200 new transcripts to test the algorithm. The transcripts were then checked by hand and it was found that the algorithm had only one misclassification, giving it a 0.5% error rate.

6.1.3 Warning Banner Finder Results

Once the algorithm had been developed, it was used to test all hosts, for which there was data, for the presence of warning banners. The first step was to filter the telnet

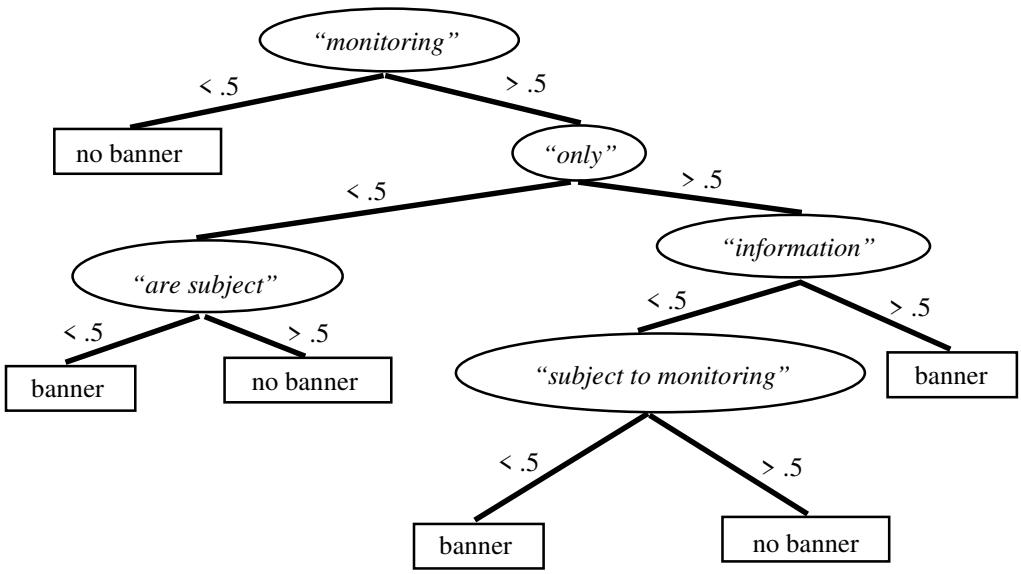

Figure 6-1: Decision Tree for Warning Banner Finder

transcripts from the 13 weeks such that, for each inside host (host on a network monitored by the baseline system), the first transcript containing a connection to that host was kept. Subsequent connections to the same host were ignored, as were all connections to machines whose name could not be resolved (and was thus listed as "unknown"). This filter found 1843 distinct hosts and saved one transcript for each of them. The warning banner finder was then run on each transcript. It found that 956 of these transcripts did not contain a valid warning banner. One caveat to these results is that some systems did not display banners until after a user successfully logged in. Thus, some hosts may have had valid warning banners but were counted otherwise because the transcript for that host had only failed login attempts. Still, these numbers indicate that running an algorithm such as this one may prove quite useful in making sure all systems are adequately protected with valid banners.

6.2 Password Checker

The password checking algorithm records a user's username and password as they log into a system and then checks the strength of their password. Its goal is to make

systems more resistant to password cracking attacks by alerting administrators to users who have weak passwords. The password checking system is actually made up of two parts: the algorithm which extracts the username and password and the algorithm which checks the strength of the password. This section describes both the password extraction and checking algorithms and then presents the results of running the Password Checker on the 13 weeks of data.

6.2.1 Password Extraction Algorithm

The password extraction algorithm works by finding usernames in the dest file and then using these strings to decide where the passwords are in the corresponding init file (recall that the init file is the file containing information from the client, which would be what the user typed, and the dest file is the file containing information from the server). Usernames are found by looking in the dest file for a login prompt. The username should then be the string between the prompt and the end of the line.

Once all usernames are found in a given dest file, passwords are extracted from the corresponding init file. Virtually all systems will have a user enter their username and then enter their password. Therefore, the password should be on the line immediately after the username. These username password pairs are then given to the password checking script (which will ignore all but the last pair).

The above procedure works quite well with clean transcripts. However, character doubling (described above in section 2.3.2.3) causes some non-trivial problems. These problems do not occur with the transcripts generated by the tcptrans utility developed as part of this thesis and described in appendix A. The first problem caused by character doubling is that a username from the dest file may not match the same username in the init file because characters may be doubled in file one and not the other. Thus, a fuzzy string compare must be used. Two strings are assumed to match if one string does not have any characters not present in the other and one string does not have more than twice as many instances of a particular character as the other.

The next problem is that the new line character can be doubled just as easily as any other character. Thus, the algorithm can not assume the password is on

the line directly below the username. If the characters are doubled singly (such as "aabb\n\n" where "\n" is the newline character), then the line after the username will contain a blank line. However, if the characters are doubled in groups (such as "ababcd\ncd\n") then the last part of the username which is in the group with the newline will appear on the next line. Sometimes the groups are so large that the entire username is doubled as a unit. In order to figure out where the password is, the algorithm must look at the line with the username, the next line (the second line) and the line after that (the third line). Figure 6-2 presents a flowchart describing the logic used to decide which line (second or third after the username) the password is on.

It was found that the entire password was doubled as a unit more often than users used their usernames as their passwords, so if the second line matched some or all of the first line, then the algorithm will assume the third line contains the password. Otherwise, the algorithm checks to see if the current username is the last in the list. If it is not, then the third line is compared with the next expected username. If they match, then the second line is assumed to be the password. If they do not match, the third line is checked to see if it is either empty or if it matches some part of the second line (indicating that the third line exists because the newline in the password was doubled). If one of these conditions is true, then the second line is assumed to be the password. Otherwise the third line is assumed to be the password. Backing up a step, if the current password is the last in the list, then the algorithm checks to see if the third line exists. If it does not, then the second line is assumed to be the password. Finally, if the second line either matches the user name from the dest file (using the fuzzy compare) or it is blank, then the third line is used, unless it is blank, in which case the second line is used.

The final problem, and the hardest to correct for, is that characters in the password may be doubled. Given that a password can, by definition, contain an arbitrary string of characters, it is almost impossible to tell if the password has been doubled or if what the algorithm recovered is what the user really typed. The only guide is that, if all the characters in the username were doubled, then the characters in the password

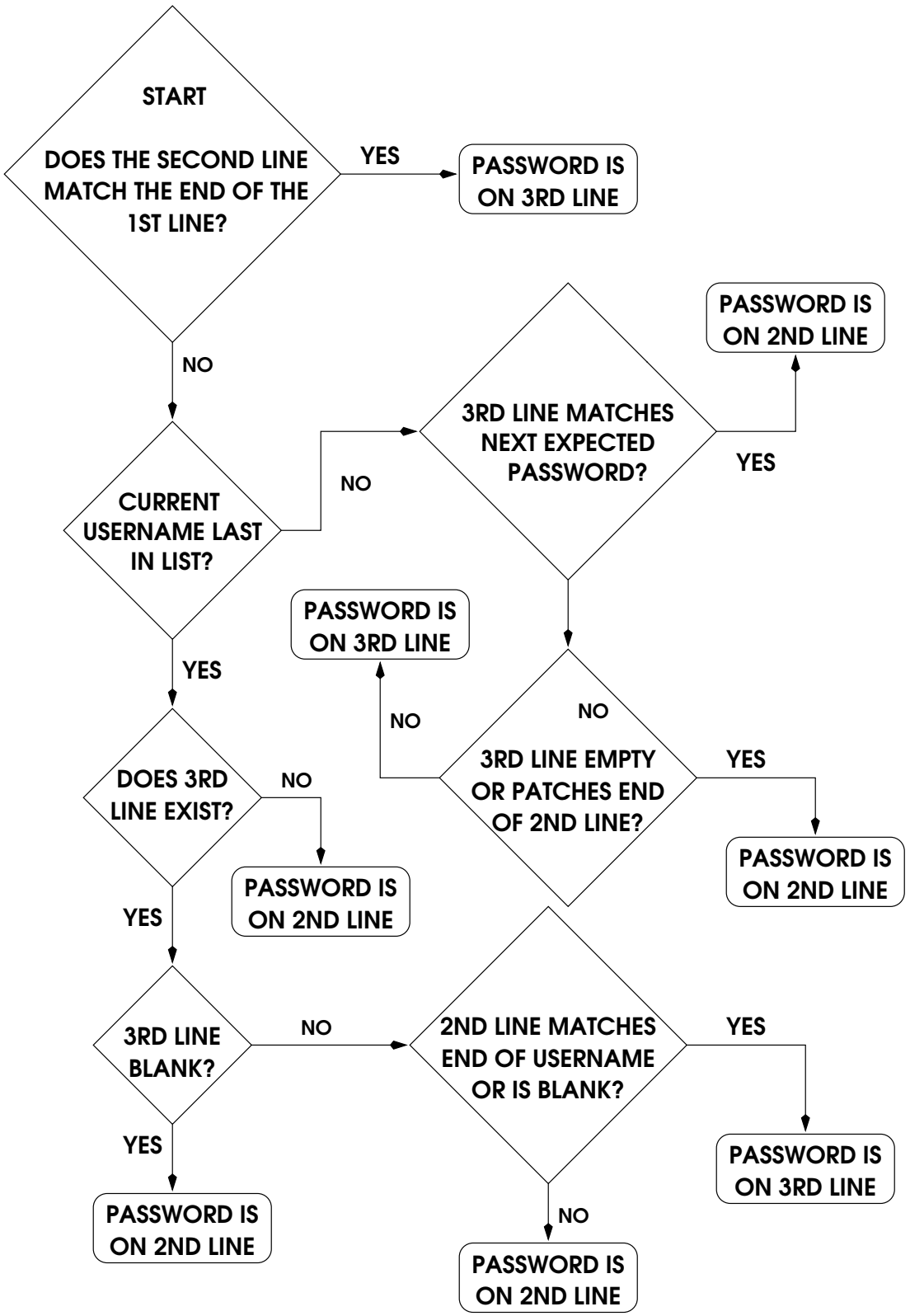

Figure 6-2: Logic Used to Locate a Password

probably are as well. Therefore, removing the second instance of all double characters is more likely to produce the correct password from an incorrect one than the other way around. One other trick used by the algorithm is that, if a user logs in more than once, the script can choose the best looking password to keep. It does this by looking at the two passwords and, if they match using the fuzzy compare, then the shorter of the two is kept and the other is thrown away (if they don't match, they are both kept and assumed to be two separate passwords).

6.2.2 Password Checking Algorithm

The password checking algorithm takes the username and password and uses these two words, in conjunction with a 50,000 word dictionary, to determine how easy the password would be to guess with a password cracking program or a dictionary attack. First, the algorithm compares the password to the username. If they match, a value of zero is returned. Next, it compares various permutations of the password against the username (see Table 6.2.2 for a description of the permutations) and returns a value of 1 if any of the permutations match. The algorithm then compares the password and permutations of the password to all the words in a 50,000 word dictionary of common words and key sequences and returns a value of 2 if the password itself matches or 3 if one of the permutations match. A value of 4 indicates that the password contains only all upper case or all lower case letters. A value of 5 indicates that the password was five characters or less in length. Finally, a value of 6 indicates that the algorithm considered this password to be reasonably strong. For example, the password "joe1" entered with the username "joe" would receive a score of 1 because "joe1" matches "joe" with the "1" removed from the end. The password "bui1d" would receive a 3 (assuming that the word "build" is in the dictionary) because, with the "1" replaced with an "l", "bui1d" matches a dictionary word.

password as is
password backwards
password with all “0” ’s replaced by “o” ’s
password with all “0” ’s replaced by “o” ’s and backwards
password with all “1” ’s replaced by “l” ’s
password with all “1” ’s replaced by “l” ’s and backwards
password with first character removed if not a letter
password with first character removed if not a letter and backwards
password with last character removed if not a letter
password with last character removed if not a letter and backwards

Table 6.1: Permutations Used By Password Checker

6.2.3 Password Checker Results

The password checking system was given a preliminary test on 198 telnet transcripts. It was able to recover passwords from 82% of them (in the others it couldn’t find a username in the dest file, it couldn’t find a match to the dest username in the init file, or the transcript contained no data). Of these 168 transcripts for which passwords were recovered, 15% of them were incorrect (either the wrong line was chosen or the password had characters doubled in it that the extraction algorithm couldn’t detect). In a separate measure, 20% of the 168 passwords were found to be weak (score of four or less).

In a full test on all 79,765 telnet transcripts, passwords were recovered from 50,575 (63%) of them and 6993 (15%) of those scored less than four. The algorithm was effective in rapidly detecting weak passwords from sniffer data yet requires much less computation than CRACK and similar programs because it has access to the clear text passwords from the sniffing data. Therefore, it does not have to use extra computation to encrypt each string to compare with the encrypted password string. Furthermore, where programs such as CRACK have rules which form a one to many mapping from a dictionary word to a number of possible passwords (such as the word with each of the ten different digits appended to it), the permutations used by the Password Checker actually map many passwords into one trial word to search the dictionary.

Chapter 7

Conclusions

Many of the algorithms discussed in this thesis showed results promising enough to warrant future investigation and inclusion in a larger intrusion detection system. Perhaps more importantly, the work done in developing these algorithms shed some light on the broader subject of building intrusion detection systems as a whole. This chapter discusses the various algorithms from the standpoint of what their potential may be and what was learned during their development.

7.1 The Password Guessing Detector

The Password Guessing Detector greatly aids in finding dictionary and doorknob rattling attacks by filtering out non-malicious connections and presenting the remaining sessions in a way which simplifies human analysis. Given that the only attack which can appear in a transcript containing just failed logins is a password-guessing attack, preprocessing the transcripts with this algorithm would allow analysts to quickly find all password guessing attacks and then ignore the 35% of the transcripts containing only failed logins. Thus, the effective false-alarm rate of the baseline system can be reduced by 35% with no modification to that system and very little additional work on the part of the analysts.

Furthermore, the algorithm is relatively simple and uses little computational overhead. Although the algorithm ran in batch mode on the previously collected tran-

scripts, its small system footprint would allow it to easily keep up with real-time data. Statistics on all connections seen in the last few weeks or months could be continuously updated as users logged in (or failed to do so). Recognizing attacks in real-time would both give administrators a chance to take action before the attack is successful and let the system gather information about the attacking host through reverse fingers or other probes while the attack is occurring.

The most important lesson learned from this algorithm is that how the data is presented is almost as important as what the data is. As mentioned in section 3.3 where the results for the password guessing detector algorithm are presented, one run through the data had the checks for password strength and repeated passwords turned off. Thus, the output included all 23,315 connections without a valid login. The difference between the Password Guessing Detector and the baseline system was that, instead of presenting them as a collection of transcripts, the connections were summarized into a sorted list. This allowed one person to check 13 weeks worth of data for large scale dictionary attacks in less than an hour. The list of all usernames tried and the total number of attempts across all connections also made scanning for the dictionary or doorknob rattling attacks very quick. These results imply that the way in which data is presented to the human analysts is often as important a determination of workload as a system's false alarm rate.

7.2 New Keyword Weighting System

By using a neural network trained on real data to weight the keywords and by modeling both legal and illegal behavior instead of just illegal behavior, this algorithm was able to increase the purity of the baseline system by almost an order of magnitude, bringing it close to an acceptable range. Furthermore, the computational overhead of the neural network is very small, requiring only a weighted sum of 40 values to score each transcript. Replacing the baseline system's current weighting scheme with this algorithm would be relatively straight forward and would immediately improve its performance.

In general, this algorithm demonstrates the need for a large body of normal user activity with which to train a system and test its false alarm rate. Without a large body of real world data, it is very difficult to say how a given weighting scheme will perform.

7.3 Shell Tracking Algorithm

The bottleneck verification idea appears promising. Based on a limited amount of data, the shell tracking algorithm demonstrated an ability to detect different kinds of attacks with no a priori knowledge about those attacks. It is conceivable that this algorithm (or a variant of it such as those discussed in section 8.3 on future improvements) could detect a wide range of attacks, including new attacks not widely known, with a low false alarm rate. This system already runs in real time, potentially allowing it to take offensive action should an intrusion be discovered, before the attacker has had time to compromise data on the victim machine. Finally, the limited number of audit records monitored by this system means that the volume of audit data which each host would need to export to a central collection machine is quite small. This fact, along with the low computational requirements of the algorithm, suggest that a central host could receive and process data from a reasonably large number of hosts in real-time without falling behind.

7.4 Intrusion Prevention Algorithms

The results from the banner finder and password checker show that simply creating security policies is not enough. Users and systems must be continuously monitored for compliance. These algorithms are simple and fast enough to run in real time and would be relative easy to include as part of a larger intrusion detection system.

Chapter 8

Future Work

While many of the algorithms described in this thesis demonstrated good performance, they all would benefit from a bit more refinement. This chapter lists the current weaknesses of each algorithm and some ways in which those weaknesses could be overcome.

8.1 Password Guessing Detector

The single greatest detriment to this algorithm's performance is the noise in the transcripts. Simplifying it to work with clean transcripts, such as those produced by the tcptrans utility described in appendix A, would allow the algorithm to recover the usernames and passwords much more accurately. Clean passwords, in turn, would keep weak passwords from being considered strong simply because they were corrupted and would allow the rule discarding transcripts with all strong passwords to be reinstated.

The lack of any dictionary attacks in the data, along with other sites reporting doorknob rattling attacks in excess of one every other day[1], suggest that a better approach to detecting these two attacks may be to have two separate algorithms. The current algorithm would be used to find dictionary attacks while another algorithm, which only looked for attempted logins with commonly attacked account names or account names not existing on the system, would be used to detect doorknob rattling

attacks. This approach would keep the reports of small doorknob rattling attacks without too many connections from getting buried in the list of connections reported by the dictionary attack detection algorithm.

Regardless of whether one or two algorithms are used, they should be implemented in real time. This would allow the system to both alert administrators to large attacks in progress and probe the attacking machine for information about the identity of the intruder while the attack is in progress.

8.2 New Keyword Weighting System

One improvement to this system would be to implement it in real-time. As with the other algorithms, real-time warnings would allow defensive measures to be taken while the intruders are still active. Furthermore, instead of totaling all keywords seen in session up to the current point, a sliding window could be implemented such that the keyword counts would only represent keywords currently in the window. This would give the algorithm an indication of which keywords occurred close to each other. The idea of context could also be added to the counts such that keywords typed by the user would be distinguished from those found in a file or as the response of a command. Finally, given the drastic improvement in performance from simply reweighting the keywords used by the baseline system, it is quite possible that the set of keywords itself is not optimal. Generating n-grams from the transcripts and using a feature selection algorithm to choose the best words or phrases may produce a set of keywords that can differentiate between attacks and normal connections much more accurately.

8.3 Bottleneck Verification Algorithm

There are two extensions to the bottleneck verification algorithm that would make it more robust and able to detect other types of attacks. The first is to recognize command shells based on their properties instead of simply their name. Command

shells are likely to behave very differently from other processes on the system (they fork off child processes quite frequently, the child processes are usually system binaries, etc.). Thus, a statistical or expert system could probably be designed to identify a process as a command shell based on the audit records it generates. Recognizing command shells by behavior would prevent intruders from hiding from the system by renaming the shell, as well as give the system the ability to detect other types of shells.

The second improvement would be somewhat more involved and is directed at finding attacks that do not launch a shell at all, but instead modify some system file to add a back door. One popular action taken by many attack scripts distributed on the Internet is to add a new user to the password file with a user ID of 0 (i.e. the user ID of root). Thus, instead of looking for root command shells as evidence of a user in the root state, the system should look for a process running as root and performing file writes to any file which the user who launched the process does not have write permission to, and which that process does not normally write to on its own. This approach involves somehow obtaining and maintaining a list of what files suid programs (those that run as root, even when launched by a user) normally write to. This list could be learned by an algorithm that observes normal activity of the system over some training period. The main advantage of this approach is that it has the potential to detect a wide variety of attacks. In some ways, this idea is similar to the specification based approach [11] and to USTAT (a UNIX-specific implementation of STAT) [7].

8.4 Password Checker

Of all the algorithms using the transcripts as input, this algorithm suffered the most from character doubling. Modifying this algorithm to use clean transcripts would allow it to extract usernames and passwords with nearly 100% accuracy. The actual checking of the strength of the passwords would be more accurate as well. Another way to improve the performance of the Password Checker so that it finds more weak

passwords would be to use a bigger dictionary. The algorithm would also benefit from additional rules for permuting the password to the system. These rules should reflect the same permutations used by password cracking programs, since it is with respect to them that the password must be secure. Finally, implementing a real-time version of the algorithm would be reasonably straight forward and would facilitate its inclusion in a larger real-time system.

Appendix A

Tcptrans Transcript Generation Utility

Tcptrans is a transcript generation utility created as part of this thesis. It produces transcripts similar to those generated by the baseline system, but without the associated character doubling. This appendix describes the current state of the utility and gives some ideas for future improvements.

A.1 Current Functionality

Tcptrans takes a file of raw packets created by tcpdump (a freely distributed network monitoring tool) and produces ASCII transcripts of all TCP connections found (see figure A-1). The transcripts generated contain a header with the source and destination IP addresses, the source and destination ports, and the start and end times of the connection. The body of the transcript contains the data found in the TCP packets (with a few modifications). It also includes a line at the beginning of the body if the connection was open before data collection was started, and a line at the end saying how the connection terminated. It can process multiple dump files at one time, treating them as one continuous stream of packets.

Two elements of the TCP data are modified or discarded in generating the transcripts. First of all, non-printable characters (ASCII values less than 32 and greater

than 126) can be either not printed or represented by their ASCII value (or a text tag in the case of common characters like DEL or ESC). The other item, which is specific to telnet connections, is the telnet control sequences. These allow the client and server to agree on what optional parts of the protocol will be used for the current session. At the present these are discarded; however, since at least one attack involves information sent in these options, there are plans to print out a summary of each option as it is encountered. Figures A-2 and A-3 show the init and dest files for a transcript generated with the tcptrans utility. This session is just like the one shown in the sample baseline system transcript (figures 2-3 and 2-4). Comparing these figures shows that the tcptrans files do not have any extraneous noise from the telnet options.

Another attribute of tcptrans, which has been found to be very useful to intrusion detection systems that use the transcripts as data, is that it can print out timing information about when the packets were received. This information can either be the time each packet was received, or the time each carriage return was received. The latter option has been used to aid in synchronizing what was typed in the init file to the response in the dest file. The per-line time stamps can also be printed in separate files, so that an intrusion detection system can still have timing information without compromising the readability of the actual transcript.

Finally, instead of generating transcripts for all the connections, tcptrans can print out a one-line summary of each connection giving the start and end times, the IP addresses and ports of the machines involved, and the number of packets sent in the connection. Figure A-4 shows an example summary file with three telnet connections between the same two machines.
Table A.1 gives the currently supported options and their meanings.

A.2 Future Plans

While it is a very useful utility, the current implementation of tcptrans has a few problems scheduled to be fixed in the next version. First of all, it keeps between

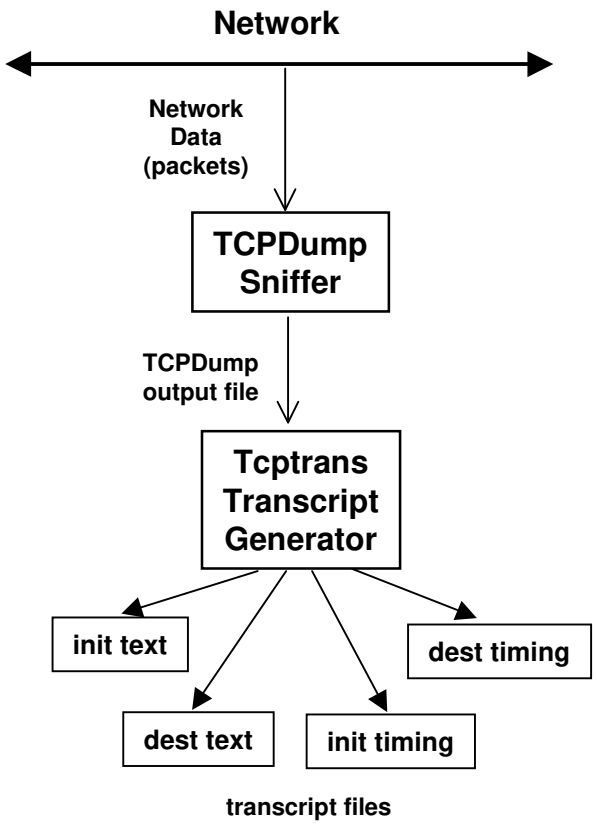

Figure A-1: The Tcptrans Transcript Generation Utility

```
Source IP: 172.16.112.10
Dest IP: 172.16.112.50
Source Port: 33691
Dest Port: 23
Start time: May  8 12:15:36 1998
End time: May  8 12:15:44 1998
------------------------
mary
mary12
ls -a
pwd
exit

[The connection closes normally at this point]
```

Figure A-2: Sample Init File Generated by the Tcptrans Utility

```
Source IP: 172.16.112.10
Dest IP: 172.16.112.50
Source Port: 33691
Dest Port: 23
Start time: May  8 12:15:36 1998
End time: May  8 12:15:44 1998
-------------------------


UNIX(r) System V Release 4.0 (pascal)

login: mary
Password:
Sun Microsystems Inc.   SunOS 5.5       Generic November 1995
mary@pascal: ls -a
./       ../      .cshrc
mary@pascal: pwd
/.sim/home/mary
mary@pascal: exit
mary@pascal: logout

[The connection closes normally at this point]
```

Figure A-3: Sample Dest File Generated by the Tcptrans Utility

```
1 05/09/1998 12:52:55 00:00:00  33697     23 172.016.112.010
      172.016.112.050 10 -
2 05/09/1998 12:52:49 00:00:00  33696     23 172.016.112.010
      172.016.112.050 10 -
3 05/09/1998 12:52:41 00:00:18  33695     23 172.016.112.010
      172.016.112.050 43 -
```

Figure A-4: Sample Summary File Generated by the Tcptrans Utility

-summary	print a summary file for all connections instead of generating the transcripts
-a	generate headers similar to those produced by the baseline system
-st <packet,line,minute,file>	specifies how to print timestamps for the source (init) file.
packet	print a timestamp for each packet
line	print a timestamp for each line
minute	print a timestamp every minute
file	print a timestamp every line in a separate file
-dt <packet,line,minute,file>	specifies how to print timestamps for the dest file. These options are the same as for the -st flag
-IP <addr>	only generate transcripts for connections containing this IP address
-port <port>	only generate transcripts for connections containing this port number
-d	specify a directory for the transcripts to be created in

Table A.1: Currently Supported Options for Tcptrans

two and four open file descriptors for each connection, depending on whether or not timing information is being printed in separate files. If a dump file has a large number of simultaneously open connections, the operating system can run out of available file descriptors. A better approach would be to buffer connection data in memory until it reaches a certain size, open the file, write out all available data, and then close the file again.

Another useful extension would be to give tcptrans the ability to parse telnet options and print out a summary instead of simply discarding them. This would allow intrusions like the corrupted system library attack discussed in section 2.1 to be discovered more easily.

Finally, protocols other than telnet could be parsed as well. Currently, telnet connections are printed by a customized printing procedure that understands the protocol and filters out the options to make transcripts more readable. The data for all other services is simply printed with the normal handling of non-printable characters. Specialized printing procedures for additional services, such as X or HTTP, however,

would make those transcripts much more readable as well, especially for services like X where a majority of the data is concerned with the mechanics of displaying the X-window and not what the user is typing or seeing.

Bibliography

[1] Steven M. Bellovin. There be dragons. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 1–16, Baltimore, MD, September 1992. The USENIX Association and the Computer Emergency Response Team (CERT).

[2] Hervé Debar, Monique Becker, and Didier Siboni. A neural netowrk component for an intrusion detection system. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 240–250, Oakland, CA, May 1990. IEEE.

[3] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. Technical report, Purdue University, West Lafayette, Indiana 47907-2004, July 1990.

[4] Simson Garfinkel and Gene Spafford. *Practical Unix & Internet Security*, chapter 8, page 227. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol CA, 95472, 2nd edition, April 1996.

[5] Thomas D. Garvey and Teresa F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, pages 405–418, October 1991.

[6] L. Todd Heiberlein, Gihan V. Dias, Karl N. Levit, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 296–304, Oakland, CA, May 1990. IEEE.

[7] Koral Ilgun. USTAT : A real-time intrusion detection system for UNIX. Master's thesis, University of California Santa Barbara, November 1992.

[8] Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[9] RFC 791 : Internet Protocol, September 1991.

[10] R. Jagannathan, Teresa Lunt, Debra Anderson, Chris Dodd, Fred Gilham, Caveh Jalai, Hal Havitz, Peter Neumann, Ann Tarnaru, and Alfonso Valdes. System design document: Next-generation intrusion detection expert system (NIDES). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California 94025, March 1993.

[11] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *1994 Computer Security Application Conference*, 1994.

[12] Linda Kukolich and Richard Lippmann. *LNKnet User's Guide*. MIT Lincoln Laboratory, April 1995.

[13] Teresa F. Lunt, Ann Tamaru, Fred Gilham, R. Jagannathan, Caveh Jalali, Peter G. Neumann, Harold S. Javitz, Alfonso Valdes, and Thomas Garvey. A real-time intrusion-detection expert system (IDES). Technical report, SRI International, February 1992.

[14] Alex Muffet. CRACK. http://www.users.dircon.co.uk/~crypto/.

[15] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, K. Levitt, C. Wee, R. Yip, D. Zerkle, and J. Hoagland. GrIDS–a graph based intrusion detection system for large networks. http://seclab.cs.ucdavis.edu/arpa/grids/welcome.html, 1996.

[16] Jake Ryan, Lin Meng-Jane, and Risto Miikkulainen. Intrusion detection with neural networks. In *Advances in Neural Information Processing Systems 10*, May 1998.

[17] Stephen E. Smaha. Haystack: An intrusion detection system. In *Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Orlando, FL, December 1988. IEEE.

[18] RFC 793 : Transmission Control Protocol, September 1991.

[19] Wietse Venema and Dan Farmer. Security analysis tool for auditing networks (SATAN). http://www.fish.com/satan/.

[20] Personal communication from Dan Wsychograd at MIT Lincoln Laboratory, June 1997.