Fall 2012

Masquerade Detection Based On UNIX Commands

Amruta Mahajan
San Jose State University

Masquerade Detection Based On UNIX Commands

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Amruta Mahajan

December 2012

The Designated Project Committee Approves the Project Titled

Masquerade Detection Based On UNIX Commands

by

Amruta Mahajan

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2012

Dr. Mark Stamp	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science

ABSTRACT

Masquerade Detection Based On UNIX Commands

by Amruta Mahajan

In this paper, we consider the problem of masquerade detection based on a UNIX system. A masquerader is an intruder who tries to remain undetected by impersonating a legitimate user. Masquerade detection is a special case of the general intrusion detection problem.

We have collected data from a large number of users. This data includes information on user commands and a variety of other aspects of user behavior that can be used to construct a profile of a given user. Hidden Markov models have been used to train user profiles, and the various attack strategies have been analyzed. The results are compared to a standard dataset that offers a more limited view of user behavior.

ACKNOWLEDGMENTS

I am grateful and take this opportunity to sincerely thank my thesis advisor, Dr. Mark Stamp, for his constant support, invaluable guidance, and encouragement. His work ethic and constant endeavor to achieve perfection have been a great source of inspiration.

I wish to extend my sincere thanks to Dr. Sami Khuri and Dr. Chris Pollett for consenting to be on my defence committee and for providing invaluable suggestions to my project without which this project would not have been successful.

I also would like to thank my husband, Neeraj Mahajan, for his support and encouragement throughout my graduation.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

A masquerade attack occurs when an attacker gains access to a legitimate user's account. Such attacks undermine basic security checks due to the rights given to users once they have been authenticated successfully. Masquerade detection [18] includes collecting information about users and creating profiles for them. These profiles can be based on a variety of information including login time, login location, session start time, session end time, CPU time, commands issued, etc. If user behavior does not match the appropriate profile, the session can be classified as a likely attack. Considerable research has been focused on masquerade detection. However, achieving high levels of accuracy remains a difficult challenge [7] [4].

To detect masquerade attacks, an Intrusion Detection System (IDS) can monitor each user and look for any malicious or unusual behavior. There are two general approaches used by IDSs: signature based detection and anomaly based detection. Signature based detection is useful in detecting the attacks for which signature is known. In addition to these attacks anomaly-based detection can be used for unknown attacks. Anomaly based detection models normal behavior for each user depending on different characteristics and is an inherently challenging task.

The Hidden Markov Model(HMM) [17] is a powerful statistical tool for modeling generative sequences that can be characterized by an underlying process generating an observable sequence. In this project we first build a logging tool to collect user-issued UNIX commands. This data set is then cleaned and preprocessed and then processed using HMM. With HMM, the profile for each user will be derived from these

corresponding UNIX command sequences and then would be used for masquerader detection.

Our goal was to create a dataset which will include only the commands executed by a UNIX user. Along with the commands, we have also collected timestamp at which the command was executed, command arguments, CPU time taken for command execution. Then we ran HMM on this data to make sure it is comparable with Schonlau et al. dataset. Schonlau dataset [16] only contains UNIX commands. So, our dataset can be used for analysis based on additional information such as timestamp, command arguments.

This paper is organized as follows: Chapter 2 presents preliminaries and limitations of previous dataset. Chapter 3 shows basic architecture of masquerade detection, data collection and its processing. Chapters 4 and 5 give the implementation details. Chapter 7 explains the results and comparison with previous dataset. Finally, Chapter 8 presents our conclusions and possible areas for future work.

CHAPTER 2

Preliminaries

This chapter discusses the previous work related to masquerade detection.

2.1 Introduction

In modern computer security field, there are many authentication techniques to restrict invasion of an intruder into a computer system. Computer Systems which solely rely on authentication based on username and passwords are prone to Masquerade attacks where an attacker uses login credentials of a legitimate user to do some malicious activity.

Masquerader is an attacker who impersonates other user. He makes use of legitimate login credentials of other user to bypass the security check and to do some malicious activity. Masquerade attacks often take place inside an organization so they are hard to detect.

One way to detect such an attack is to use IDS (Intrusion Detection System) based on user profiles. These types of detection systems are called Anomaly based detection systems. Masquerade detection is a process of collecting information about all the users and developing profile for each user [7]. This information may include login time, session start time, session end time, CPU time, UNIX commands and its arguments. User profiles are specific to individual users and are created using such user information. This paper focuses on creating user profiles based on UNIX commands issued by an individual user. Profiles for each user are pre-computed based on all the UNIX commands fired by the user and are stored somewhere in the system.

Each time when the user logs into the system, these profiles are created on the fly and compared with the existing ones stored in system. Any significant diversion from normal profile (behavior) can trigger the alarm for malicious activity. There are many intrusion detection techniques present to detect masquerade attacks. However, these attacks need further attention as the rate of false positives is high during this process [13].

2.2 Masquerade detection

Some well known intrusion detection systems to detect a masquerade attack are [18], [9]. These can be grouped into two categories namely Signature-based detection and Anomaly-based detection [18]. In signature-based detection, user behavior is matched with some patterns which are known to be attack patterns. If the match occurs, it can be concluded that an attack has occurred. This type of techniques can only be used to detect known attacks not the new unseen attacks [18].

In Anomaly-based detection, a normal user behavior is captured and a user profile is created. Next time when the user logs in, his behavior is analyzed and compared with the normal one. If there is deviation beyond certain threshold percentage, red flag can be raised. In this project, we have focused on Anomaly-based detection in UNIX environment. We can assume that each user will have their own unique style to issue UNIX commands.

2.3 Machine Learning Techniques

Machine learning algorithms are widely used in the field of computer science, bio-informatics, mathematics and many more. Machine learning can be used to build a system that can learn from previous experience and use this experience to give

some feedback for unknown cases. In addition to simply writing a computer program to solve a particular problem, we need to have great amount of example data or experience to learn from.

One example where machine learning could be used is in recognition of characters of a given language written by humans. Humans seemingly are able to distinguish between different characters written say in English language but it is difficult for us to explain how we achieve it. The characters could have been written by many different people in their own handwriting and style but somehow we are able to recognize them. Using machine learning, an algorithm can be trained by giving it samples of written text and it could extract important features from that text. We further talk about two types of machine learning algorithms, that is, supervised and unsupervised learning.

2.3.1 Supervised Learning

Suppose we want to classify our input data into two categories viz. good user and bad user (masquerader) and we have some previously known example data for both the categories. The supervised learning algorithms can be used to extract important information from this known example data of both good and bad users. So in this case, one has sufficient amount of data so as to train the model for each category. Once we have the model, all the unknown users can be categorized into known categories.

Maximum likelihood is one of the most commonly used classifiers. For each user, the probability that the user is a member of that class is calculated. Higher the probability, greater is the chance of a user getting assigned to that particular category. The advantage of maximum likelihood classifier is that it achieves good separation of classes. The only disadvantage of this classifier is that it requires strong

training set to accurately describe structure of classes.

Hidden Markov Model, that is, HMM belongs to this particular learning type which is described in details in following section. We will be using this technique for categorizing good users and masqueraders.

2.3.2 Unsupervised Learning

In unsupervised learning we do not define classes beforehand and instead the data is divided into clusters with the best possible separation. Later, class names are assigned to those clusters. The advantage of unsupervised learning is that it does not need any previous example data for all the classes, that is, we do not need a large training set. The disadvantage is that the clusters formed by an unsupervised learning algorithm may not match our perception of the important classes. In this research we are not focussing on any unsupervised learning algorithm and would be purely focussing on supervised learning technique.

2.4 Hidden Markov Model (HMM)

We will refer to [17] paper in order to explain all the HMM concepts. HMM is a supervised machine learning technique. It is a probabilistic state machine where the transitions between states have fixed probabilities and are only dependant on the current state. The sequential data can be modeled based solely on the current state [17]. Like any other supervised machine learning algorithms HMM has two phases namely training and testing. HMM is trained with the known training data [17].

As oppose to a typical Markov chain where the states are completely observed, in HMM the states are never observed as they are "hidden" [17]. The model will choose the sequence of states that jointly maximizes the probability of the entire observation

sequence [17].

Once the model is constructed, we can test its response using the unknown data [17]. Higher probability represents greater similarity between observation sequence and training sequence. Lower probability results in a non-match.

2.4.1 Notation

The following information is taken from [17]. Table 1 shows the notations used to represent the HMM λ. HMM is defined as $\lambda = (A, B, \pi)$ where A is the transition probability matrix, B is the observation probability matrix which gives likelihood of observation given the state and π is the initial state distribution [17].

Table 1: HMM Notations [17]

Symbol	Description
T	The length of the observation sequence
N	The number of states in the model
M	The number of distinct observation symbols
$\mathcal{O}$	The observation sequence $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$
Q	The set of states of the Markov process
V	The set of observation symbols
A	The state transition probability matrix
B	The observation probability matrix
π	The initial state distribution
λ	The hidden Markov model, defined by its parameters A, B, and π, is denoted as λ = (A, B, π)

Figure 1 shows a generic hidden markov model where the X_i represent the hidden state sequence. The A matrix and the current state determine the Markov process which is hidden behind the dotted line. The hidden states of the Markov process, $\mathcal{O}_i$ can be observed only via the matrix B [17], which describes the probability of

observing a particular symbol in a given state. As we do not know what the transitions should be they are placed above the dotted line [17].

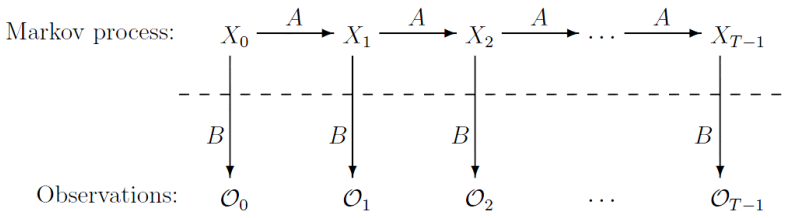


Figure 1: Hidden Markov Model [17]

The three fundamental problems [17] those can be solved by Hidden Markov Models are briefly described here.

Problem 1. Given the model $\lambda = (A, B, \pi)$ and a sequence of observations $\mathcal{O}$, find $\mathrm{P}(\mathcal{O} \,|\, \lambda)$. Here the given sequence is scored to see how well it fits into the given model λ.

Problem 2. Given the model $\lambda = (A, B, \pi)$ and a sequence of observations $\mathcal{O}$, find optimal state sequence. Here we uncover the hidden part of the Hidden Markov Model.

Problem 3. Given observation sequence $\mathcal{O}$ and dimensions N & M, find the model $\lambda = (A, B, \pi)$ that maximizes the probability of $\mathcal{O}$. Here we train the model to best fit the observed data [17].

The notations used are:

N = number of states in model λ

M = number of observation symbols

A = state transition probabilities

B = observation probability matrix

π = initial state distribution

$\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$ = observation sequence

The goal of this project is to categorize a user as either a good user or a masquerader. We are assuming that every user has an unique sequence of UNIX commands which represents that particular user. All other users which do not posses similar sequence of commands are considered as masqueraders. We propose to use HMM to differentiate between good users and masqueraders. Here the observation will be the list of UNIX commands associated with each user. The trained model is then supposed to assign high probability for commands belonging to good user and low probability to the ones in masquerade category.

2.4.2 HMM with UNIX Commands

HMM construction requires large training data. There are few publicly available data sets which provide such training data along with testing data. One such data set namely Schonlau dataset [16] is discussed in the next section. Using this data set's training data HMM can be trained (Problem 3). Then the HMM can be tested based on provided testing data (Problem 1). A high probability score indicates that both the training data and testing data have similar characteristics [18]. Whereas, low probability indicates that there is a significant difference between the two. The later is useful to recognize intrusion data.

2.5 Schonlau dataset

Schonlau et al. [16] has created a data set for comparing different masquerade detection techniques. This data set is a collection of UNIX commands captured over a period of time. It is also called as SEA data set (Schonlau Et Al.) and can be used

for training and testing purposes. The data set is publicly available for download at `http://www.schonlau.net/intrusion.html` [16]. We chose this dataset as it is publicly available and has good amount of data. There is another publicly available dataset but it has data for only four users.

Figure 2 below shows the general structure of SEA data set. The data consists of 50 files for 50 different users. Each file contains 15,000 commands. These commands are collected using acct package [19]. The first 5000 commands from each file are the commands specific to that particular user and it does not contain any masquerader's data. These are used for training purpose. Next 10,000 commands of each user are divided into 100 blocks of 100 commands. These are seeded with masquerader commands i.e. the commands other than these 50 users. These 10,000 commands can be used for testing purpose.

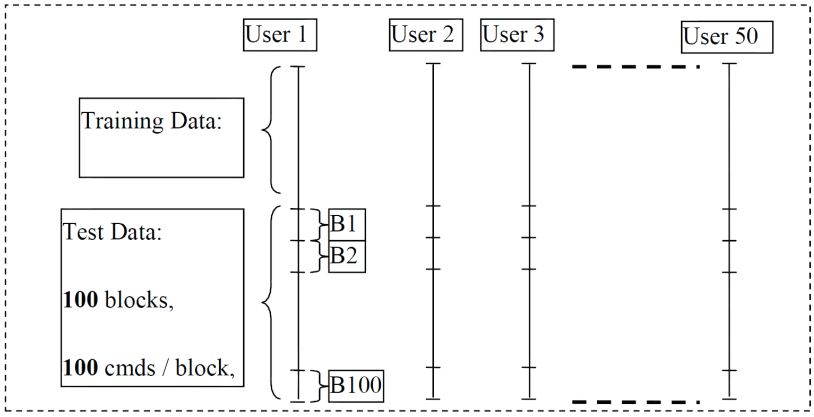


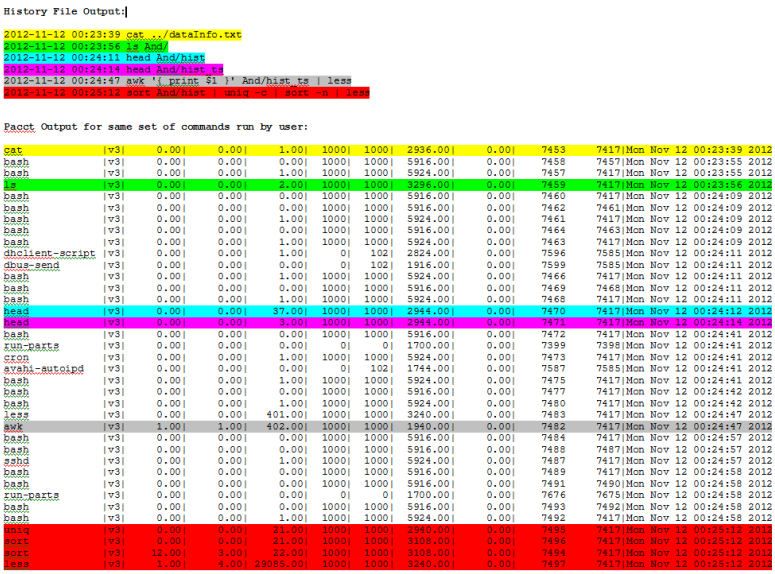
Figure 2: Schonlau data set [18]

2.6 Schonlau Dataset limitations

Of all the available fields of audit data provided by process accounting tool (acct) only the commands were considered in Schonlau dataset. Due to the way this audit tool collects audit data from the system, it is impossible to differentiate

commands executed by users from those run automatically by a shell script or by the operating system [4]. This may not give an accurate representation of the user behavior as there will be commands executed by the operating system or other utilities frequently which might or might not be common for other users depending on the operating environment.

The data is only good for training purpose. A real time masquerader data is not available so it needs to be created manually by injecting non-self data. But this approach may not be accurate and will make the detection harder. Figure 3 explains the difference between two command collection types i.e. shell history and pacct accounting history. As we can see, there are number of extra commands run by operating system in addition to commands run by user.



History File Output:

```
2012-11-12 00:23:39 cat .../dataInfo.txt
2012-11-12 00:23:56 ls And/
2012-11-12 00:24:11 head And/hist
2012-11-12 00:24:14 head And/hist_ts
2012-11-12 00:24:47 awk '{ print $1 }' And/hist_ts | less
2012-11-12 00:25:12 sort And/hist | uniq -c | sort -n | less
```

Pacct Output for same set of commands run by user:

cat	|v3|	0.00|	0.00|	1.00|	1000|	1000|	2936.00|	0.00|	7453	7417|Mon Nov 12 00:23:39 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7458	7457|Mon Nov 12 00:23:55 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7457	7417|Mon Nov 12 00:23:55 2012
ls	|v3|	0.00|	0.00|	2.00|	1000|	1000|	3296.00|	0.00|	7459	7417|Mon Nov 12 00:23:56 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7460	7417|Mon Nov 12 00:24:09 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7462	7461|Mon Nov 12 00:24:09 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7461	7417|Mon Nov 12 00:24:09 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7464	7463|Mon Nov 12 00:24:09 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7463	7417|Mon Nov 12 00:24:09 2012
dhclient-script	|v3|	0.00|	0.00|	1.00|	0|	102|	2824.00|	0.00|	7596	7585|Mon Nov 12 00:24:11 2012
dbus-send	|v3|	0.00|	0.00|	0.00|	0|	102|	1916.00|	0.00|	7599	7585|Mon Nov 12 00:24:11 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7466	7417|Mon Nov 12 00:24:11 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7469	7468|Mon Nov 12 00:24:11 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7468	7417|Mon Nov 12 00:24:11 2012
head	|v3|	0.00|	0.00|	37.00|	1000|	1000|	2944.00|	0.00|	7470	7417|Mon Nov 12 00:24:12 2012
head	|v3|	0.00|	0.00|	3.00|	1000|	1000|	2944.00|	0.00|	7471	7417|Mon Nov 12 00:24:14 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7472	7417|Mon Nov 12 00:24:41 2012
run-parts	|v3|	0.00|	0.00|	0.00|	0|	0|	1700.00|	0.00|	7399	7398|Mon Nov 12 00:24:41 2012
cron	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7473	7417|Mon Nov 12 00:24:41 2012
avahi-autoipd	|v3|	0.00|	0.00|	0.00|	0|	102|	1744.00|	0.00|	7587	7585|Mon Nov 12 00:24:41 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7475	7417|Mon Nov 12 00:24:41 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7477	7417|Mon Nov 12 00:24:42 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7480	7417|Mon Nov 12 00:24:42 2012
less	|v3|	0.00|	0.00|	401.00|	1000|	1000|	3240.00|	0.00|	7483	7417|Mon Nov 12 00:24:47 2012
awk	|v3|	1.00|	1.00|	402.00|	1000|	1000|	1940.00|	0.00|	7482	7417|Mon Nov 12 00:24:47 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7484	7417|Mon Nov 12 00:24:57 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7488	7487|Mon Nov 12 00:24:57 2012
sshd	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7487	7417|Mon Nov 12 00:24:57 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7489	7417|Mon Nov 12 00:24:58 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7491	7490|Mon Nov 12 00:24:58 2012
run-parts	|v3|	0.00|	0.00|	0.00|	0|	0|	1700.00|	0.00|	7676	7675|Mon Nov 12 00:24:58 2012
bash	|v3|	0.00|	0.00|	0.00|	1000|	1000|	5916.00|	0.00|	7493	7492|Mon Nov 12 00:24:58 2012
bash	|v3|	0.00|	0.00|	1.00|	1000|	1000|	5924.00|	0.00|	7492	7417|Mon Nov 12 00:24:58 2012
uniq	|v3|	0.00|	0.00|	21.00|	1000|	1000|	2940.00|	0.00|	7495	7417|Mon Nov 12 00:25:12 2012
sort	|v3|	0.00|	0.00|	21.00|	1000|	1000|	3108.00|	0.00|	7496	7417|Mon Nov 12 00:25:12 2012
sort	|v3|	12.00|	3.00|	22.00|	1000|	1000|	3108.00|	0.00|	7494	7417|Mon Nov 12 00:25:12 2012
less	|v3|	1.00|	4.00|	29085.00|	1000|	1000|	3240.00|	0.00|	7497	7417|Mon Nov 12 00:25:12 2012

Figure 3: History file and pacct data for same time period

CHAPTER 3

Architecture of Masquerade Detection

3.1 Basic architecture

Figure 4 shows a basic architecture for a masquerade detection technique. A good user's normal behavior is captured in terms of their previous UNIX commands as shown in Figure 2. We are assuming that each user has specific unique set of command sequence. We can say each user has his own style of using commands in his day to day life. A good model should retain all the properties of a good user. Once the profile is ready, user's ongoing commands are tested against this historic data model. If the two profiles are similar, we can conclude that the new set of commands are from the same user. If the two profiles differ, we raise an alarm saying there is something wrong and we might have encountered a masquerader.

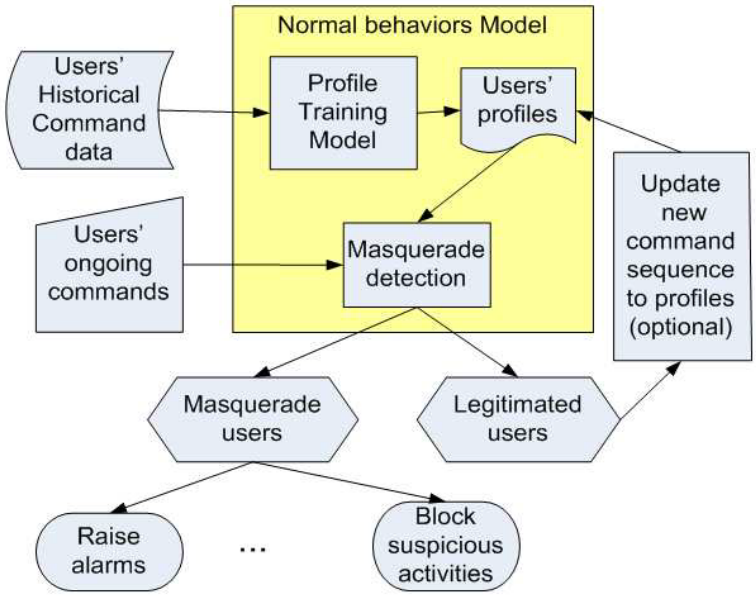


Figure 4: Architecture of masquerade detection [18]

3.2 Project road map

As we have mentioned above there are experiments done on masquerade detection techniques using previously available Schonlau dataset [18]. Though the dataset has large amount of data collected over a period of fair amount of time, these commands are collected with the help of UNIX acct auditing mechanism. Due to this the data set contains commands which are combination of commands executed by the user itself and the commands executed by the operating system but are not explicitly run by the user. For example, a shell file is a file that contains multiple commands, and running a shell file will cause all of its commands to be executed. So all of these commands would also be recorded by acct.

This project is focussed on collecting just the UNIX commands issued by an user and not recorded by the acct and to see if we could get better results with this new dataset. Also in earlier experiments additional information like command timestamp and command arguments is not considered. We tried to collect all this information from users. Data collection part is explained in the following sections.

3.3 UNIX Command logging tool

As described in the previous chapter, there are few limitations to SEA dataset which prevent us from getting accurate results. Our first goal in this project is to overcome this limitation and develop a logging tool which when installed on user's machine will collect the UNIX commands along with its arguments and other information. Other information includes session start time, session end time, CPU time, username. The later goal is to use this dataset for masquerade detection techniques.

3.3.1 Implementation

For collecting UNIX commands and additional information mentioned in section 3.1, we have used two resources [3]. First one is bash history files which are automatically generated on a Linux based system. Second one is process accounting which is achieved by using an audit tool called `pacct`. Both of these tools are discussed in details in the following sections.

3.3.2 History Files

In order to collect all the UNIX commands issued by a user, we have collected bash history files [10] which are located on user's system and are maintained by the shell. UNIX stores history for each user on the system. For bash shell this file is called `.bash_history` and is present in user's home directory. These history files are collected using a shell script [21]. Few settings are done through the shell script to set the output of these files in required format.

Settings done:

To store the timestamp along with the commands following setting needs to be done for bash shell [15]:

```
export HISTTIMEFORMAT = ‘‘%F %T’’
```

Default maximum number of commands hold by history file is 500. There are two parameters namely `HISTFILESIZE` and `HISTSIZE` present in file /etc/profile which determines the maximum number of commands that `.bash_history` file can hold. User can change this size by modifying the shell /etc/profile. This value is set to 20,000 for our purpose of collecting UNIX commands over a large period of time as

follows:

```
export HISTFILESIZE = 20000
export HISTSIZE = 20000
```

3.3.3 Process accounting by pacct

In addition to timestamps, UNIX can log all the commands run by the computer. This special type of logging is referred to as process accounting [8]. Process accounting is generally used where users are billed for the amount of CPU time that they consume. This command can also be used by administrators to check if any particular software is used by any employee that he wishes to remove from all the systems.

For this project the main purpose of using this utility is to collect CPU time and other additional information as these are not available with the shell history files. The assumption is that the log file is not deleted. The package is installed via shell script and the `pacct` log files are collected on daily basis using a `crontab` [22]. The log files are typically stored under `/var/log/account`. The location may vary depending on the Operating System.

History files output looks like:

#1336848246
tar cvfz namya-201204081536-history.tgz
#1336848732
lastcomm
#1336852385
ls

#1336852411

cat home/nm/amruta/cs297/$history\_files/.bash\_history$.3

#1336852549

pwd

#1336852615

echo $HISTTIMEFORMAT

Where, number after # is the execution epoch time for the command following it.

pacct log file output looks like:

```
acct |v3| 0.00| 0.00| 15.00| 0| 0| 172.00| 0.00| 387 386| T1

find |v3| 0.00| 0.00| 00.00| 0| 0| 168.00| 0.00| 391 390| T2

rm   |v3| 0.00| 0.00| 00.00| 0| 0| 132.00| 0.00| 398 397| T3

pwd  |v3| 0.00| 0.00| 30.00| 0| 0| 160.00| 0.00| 397 395| T4
```

Where, Tn represents time as following:

$T1$ = Tue Apr 3 07:35:01 2012

$T2$ = Tue Apr 3 07:35:02 2012

$T3$ = Tue Apr 3 07:35:02 2012

$T4$ = Tue Apr 3 07:35:02 2012

Fields in above output represents: command name, version, user time, system time, effective time, uid, gid, memory, io, time. User time, system time and effective times are ticks per second. The time field shows the start time of the process. We had requested many users to install this tool on their machines and after few months send the collected data.

3.4 Data cleaning

The data shared by the users had to be cleaned as it sometimes had garbage characters or for some commands the timestamp was missing or there were too many command line arguments. The user history files were converted to a format which was compatible with the HMM program. The processing of these history files involved several steps as below:

- Each of these files was read in line by line to check the validity of each line.
- Blank lines, extra spaces were skipped.
- If a line was solely composed of any symbolic character (meaning that it starts with a semicolon, dot, hash), it was promptly removed.
- Any line that was thought not to contain a valid command was removed. For example, sometimes user had typed 'l' instead of possibly a 'ls' command. Or there is series of same character typed by mistake as 'tttttttt'.
- All the commands that started with sudo were converted to sudo_. Meaning 'sudo mkdir' would be converted to 'sudo_mkdir'.

CHAPTER 4

Project Implementation

This section all the implementation details for our project.

4.1 Training HMM

We use UNIX commands collected from different users as `DataSet` for training HMM. We have limited amount of data for this research and we want to use it efficiently. To get meaningful results from this relatively smaller dataset, cross-validation techniques are usually used. Following section gives a brief overview of k-fold cross-validation.

4.1.1 K-fold Cross Validation

The basic idea is to efficiently divide the entire dataset into training and testing datasets such that they do not intersect. Then we run the experiment number of times on different subsets so as to increase the efficiency of performance evaluation. Usually greater part of the dataset is used for the training purpose and a small fraction is kept aside for the testing purpose.

In each round of k fold cross validation, one set will be used as a testing set and all remaining sets will be treated as a training data. In order to get accurate results, multiple rounds of cross validation are performed using a different set at each round.

We used five fold cross validation for our project. Each round of this process is referred to as a "fold". Here four sets are used as training data and one set is used for testing. We trained our model using UNIX command sequence for a

particular good user. We extracted these sequences of UNIX commands from each user's history file. We combined all the command sequences into one long observation sequence [23]. The process was repeated five times. For every iteration we changed the test data subset and finally got five HMM models per user. We can observe that the scores for the test data files belonging to a good user for which we trained the model are higher while those for the other user files are low as all the other users are considered as masqueraders. We take average of the highest score amongst other user's (masquerader's) scores from test data and highest score from good user's scores as a threshold. Figure 5 shows the training and testing phase for User 0.

4.1.2 Evaluation

We use comparison set for testing phase. Comparison set comprises of two parts as shown in Figure 5. First part is the set of the good user for which we trained the model and which is not been used in the training phase. This part will be used for validation purpose, that is to see how good our model performs. Second part includes all the sets from remaining users which we consider as masqueraders.

In the testing phase, we score the set of good user which we kept aside in the training phase. In Figure 5 User0Set0 is such a set which was kept aside for validation purpose during five fold cross validation. If we see the score for this particular set its -4.457 which is much higher than other test files (which belong to other users considered as masqueraders). We can say that higher score indicates a match, that is the set belongs to a good user. Lower scores correspond to masqueraders.

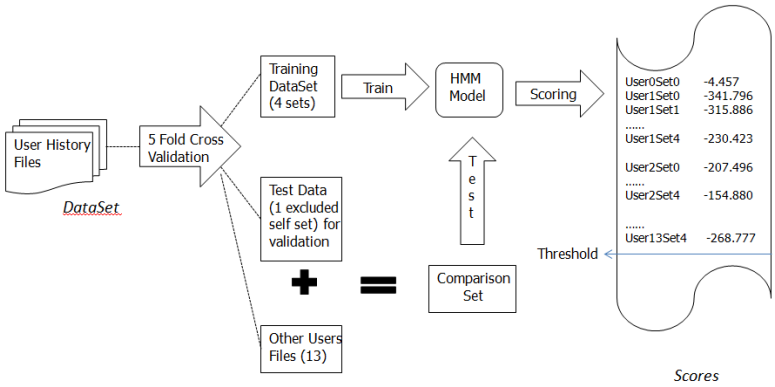


Figure 5: HMM training

CHAPTER 5

Experimental Setup

5.1 Data Processing

The data set contains all the files used for training and testing purpose. Original user history files which we have collected from different users is first processed to have them in proper format. Then this file is split into five equal parts so that we could use it for 5-fold cross-validation. We have written a perl script to do this task. Each file in this folder has name `USER[0-N]SET[0-4]`. Where N = total number of users. This folder will have 5 sets for each user. So,

Total number of files in data set = Total number of users * 5

For each data set file a corresponding .in file is created. First line of .in file represents 'T' that is total number of observations for that file. Remaining lines represent the unique index number given to each command in that file. So at the end of this process, we will be having a long sequence of indices corresponding to their respective commands.

There are also .alphabet files. First line of this file represents number of unique observation symbols in that file which we denote as 'M'. Remaining lines will have unique command names on each line.

Once the training phase is over, models are created for all sets of each user. So we will have 5 models per user. Table 2 explains about file `4_USER0_N2_E0.model`.

After testing phase, score files for all sets of each user are generated in Result folder. The name of file will look like `4_USER0_N2_E0.score` where all the symbol meanings remain same as above. Each score file has scores for two sections: 'Files of

Table 2: Model file name meaning

Value	Meaning
4	number of sets used in training phase
USER0	user number
N2	number of HMM states = 2
E0	set number which was excluded in training phase and used for validation = 0

the same user' and 'Files of other users'. A typical score file looks like below:

Result/4_USER0_N2_E0.score

Files of the same user:

USER0SET0 -4.457

Files of other users:

USER1SET0 -341.796

USER1SET1 -315.886

USER1SET2 -319.450

USER1SET3 -406.268

USER1SET4 -230.424

USER2SET0 -207.496

USER2SET1 -59.990

USER2SET2 -79.644

USER2SET3 -105.259

USER2SET4 -154.880

'Files from same user' section in this score file has the validation scores. This is the reason the scores for this section is high as the validation data belongs to a good user. 'Files from other users' section contains scores for all the remaining users (masqueraders). As we can see there is a particular range of scores for each user, that is, USER 1 scores range from -230 to -406 whereas USER 2 scores are in different range.

CHAPTER 6

Accuracy Measure

6.1 Possible Outcomes

We have four possible outcomes as true positive (TP), true negative (TN), false positive (FP) or false negative (FN). Whenever a masquerader is correctly classified as a masquerader, we say its a true positive. We get a true negative when a good user is correctly classified as a good user itself. Ideally our masquerade detection technique should give only TPs and TNs. But in reality we might get a lot of false positives and false negatives. A false positive occurs when a good user is misclassified as a masquerader. A false negative occurs when a masquerader is misclassified as a good user instead of a masquerader and passes the check. Ideally we would like to keep both of these down to zero but usually only one of them can be kept low. In masquerade detection though we want to have low false negative rate as we don't want any masquerader to get system access as a good user. An ideal masquerade detection technique should be capable of detecting all the true positives and should have zero false positives. Figure 6 shows all possible outcomes as discussed.

6.2 Receiver Operating Characteristic

Originally, the receiver operating characteristic (ROC) was developed for signal detection applications [6]. ROC is widely used for evaluating the performance of various machine learning algorithms [5], [2]. On X-axis, we have false positive rate(FPR) i.e. when a good user is identified as a masquerader and Y-axis has true positive rate (TPR) i.e. when a masquerader is correctly identified [20]. TPR and FPR are calculated as below:

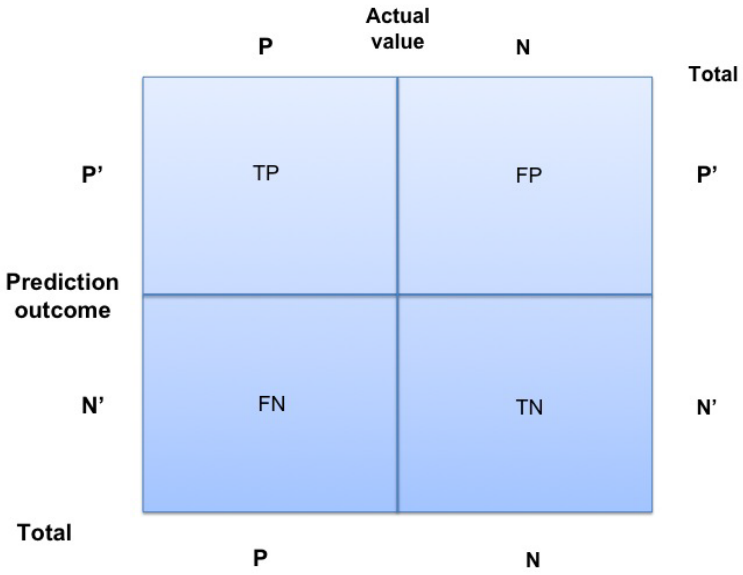


Figure 6: Possible Outcomes [11]

$$TPR = \frac{TP}{TP+FN}$$

$$FPR = \frac{FP}{FP+TN}$$

Figure 7 shows example of a ROC curve.

The top left portion of a ROC curve represents higher true positive rate. So if the curve lies in that part, we get the better classification. Diagonal is our reference line. An algorithm with 50% accuracy will have its ROC curve along the diagonal line [20]. An ideal case is shown by the brown line with the diamonds from Figure 7. It has 100% true positive rate and 0% false positive rate. The blue line having squares on it represents an algorithm with approximately 78% true positive rate and 10% false positive rate [20]. The line with triangles on it shows that this algorithm is not performing that well. The area under the curve (AUC) shows how well a technique can distinguish between two groups, in our case a masquerader versus a good user. In ROC analysis, if the area is closer to 1.0 then the result of the test is considered to

be better compared to when the area is closer to 0.5. We will be using ROC curves for our HMM detector.

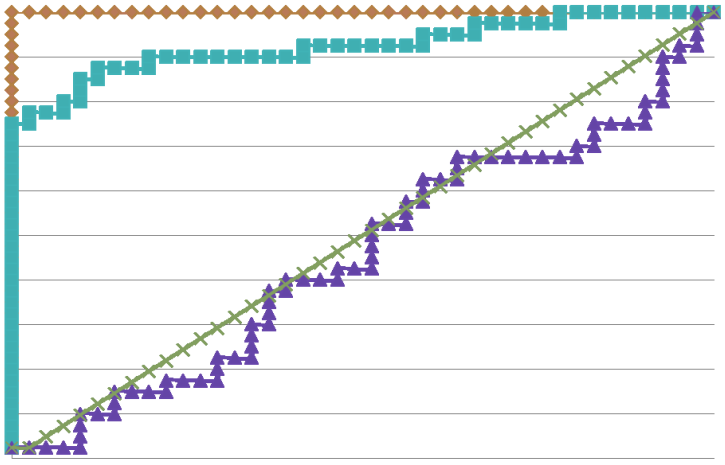


Figure 7: ROC Example [20]

CHAPTER 7

Experimental Results

We have considered different number of states for the Hidden Markov Model i.e. $N = 2to6$. We ran our experiments on all these states. As we have used five fold cross validation, after testing phase we got five models for each user. We plotted graphs for all the users. For subsequent amount (half) of users, there was a complete separation between good users' scores and all the masqueraders' scores. This means that log likelihood per command (LLPC) for all the good users were higher than those of all the masqueraders. For the remaining models, we observed some overlapping between the two scores. This happened because some masqueraders had higher scores than a good user's score.

Let's look at the result of running HMM for one of the good users i.e. 'User 10'. Figure 8 shows the result for this particular user. We used $N = 2$ i.e. the HMM with two number of states. It can be observed from the graph that all the masqueraders have lower scores than that of the good user (User 10 in this case). The result is not perfect though as two masqueraders have been misclassified as good users. Also one good user's set is been misclassified as a masquerader.

For each model, we found a number of false positives as well as false negatives. When a good user is misclassified as a masquerader a false positive occurs . So for this particular model, we have two false negatives and one false positive.

Hidden Markov Model classifies a user into either category by comparing its score with the threshold value. Therefore setting our threshold value to some higher number or a lower number affects our results. Depending on the threshold value, number of

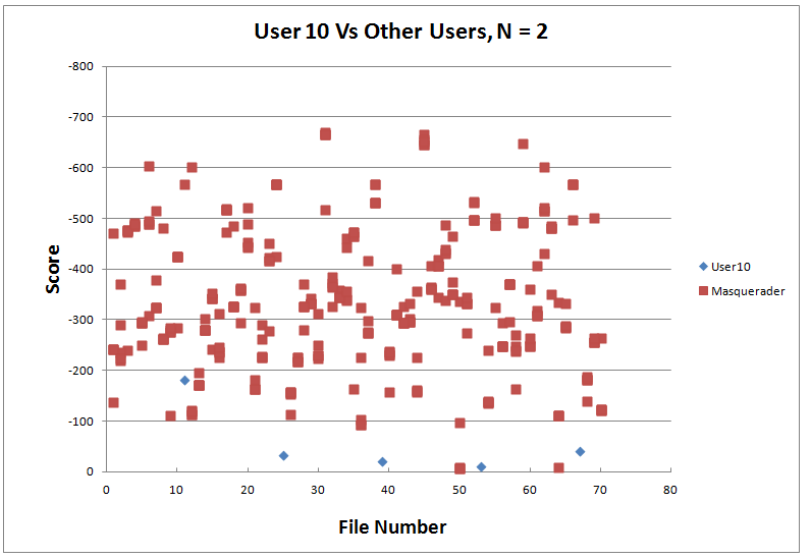


Figure 8: Log likelihood per command of User 10 Vs masqueraders for N = 2

false positives and false negatives may vary. Higher the threshold, fewer the false positives. However setting the threshold higher usually results in more number of false negatives as more users may have scores lower than the threshold.

We have plotted a Receiver Operating Characteristic (ROC) Curve for the above model considering different threshold values. Figure 9 shows the ROC curve for above case. Area under the curve for this ROC is 0.963 which is much higher than 0.5 but this case is not as good as other cases.

Next, we see some of the typical cases where there is a good separation between a good user and all the masqueraders. Figure 10, Figure 11 and Figure 12 shows such users.

As we can see for all the above users there is a complete separation of two categories. The ROC plots for these users are shown in Figure 13, Figure 14 and Figure 15 respectively. AUC for User 2 and 4 are 1.0 which are ideal cases as there are no false positive or false negatives. But for User 0, AUC is 0.958 as 1 of the

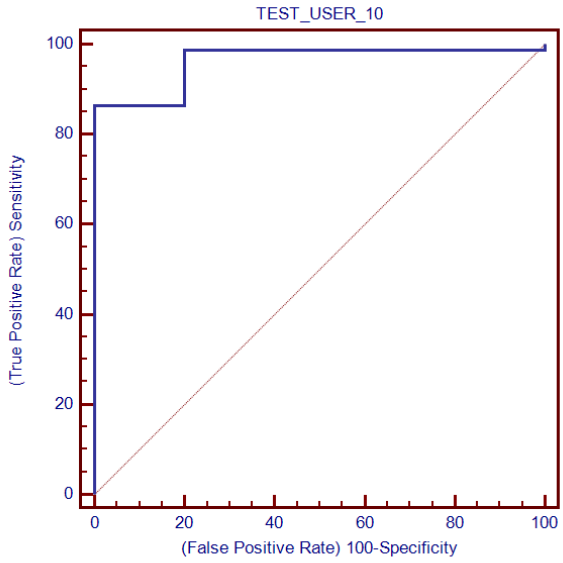


Figure 9: Receiver Operating Characteristic (ROC) Curve, User 10, $N = 2$

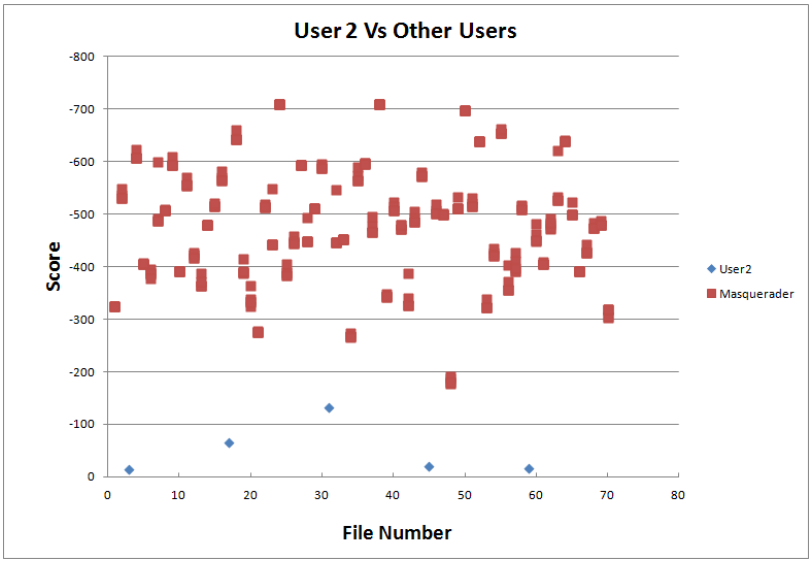


Figure 10: Log likelihood per command (LLPC) of good user 2, masqueraders for $N = 2$

masquerader is very close to a good user range.

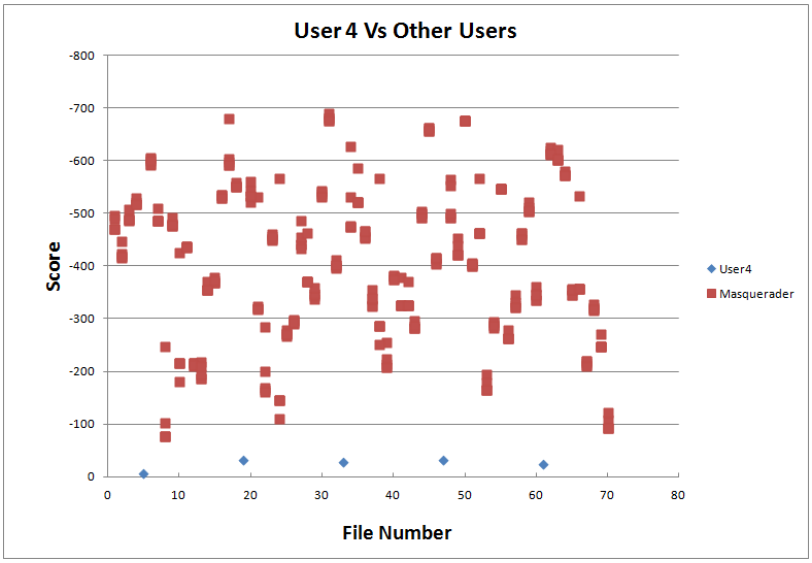


Figure 11: Log likelihood per command (LLPC) of good user 4, masqueraders for $N = 2$

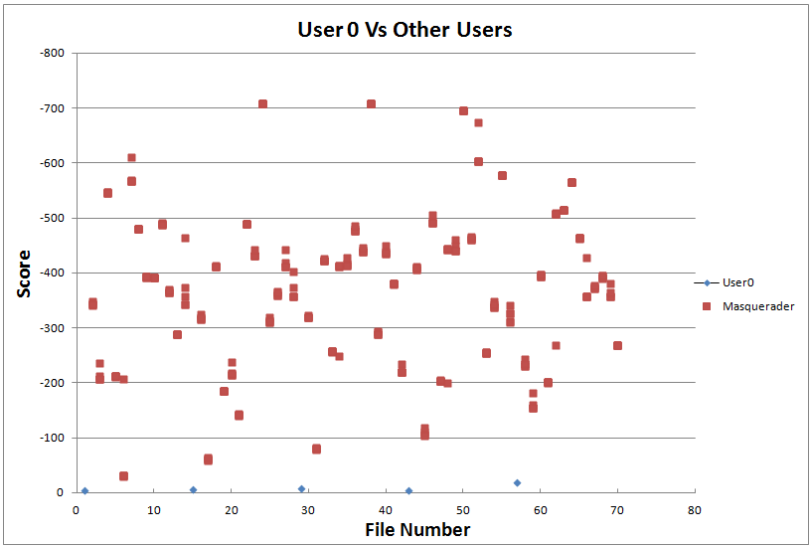


Figure 12: Log likelihood per command (LLPC) of good user 0, masqueraders for $N = 2$

7.1 Multiple HMM states

We also ran all the test cases for $N = 3, 4, 5, 6$ but there is not much significant change in results for all these states. So we can say that HMM states does not affect

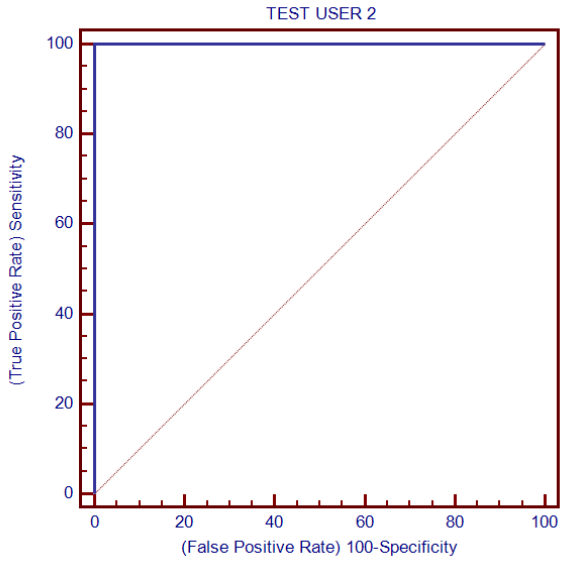


Figure 13: Receiver Operating Characteristic (ROC) Curve, User 2, $N = 2$

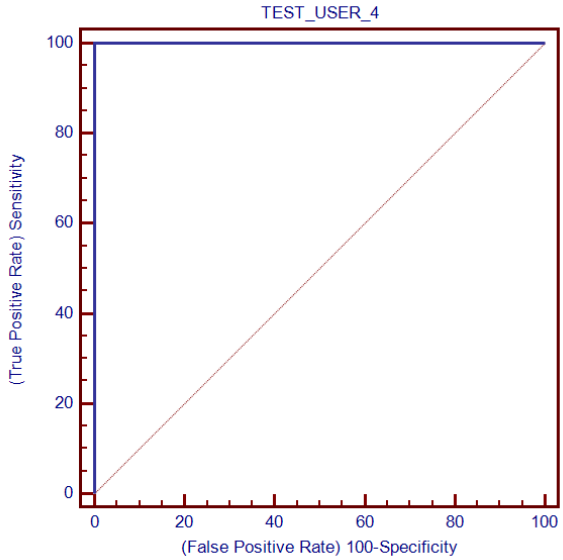


Figure 14: Receiver Operating Characteristic (ROC) Curve, User 4, $N = 2$

our results much.

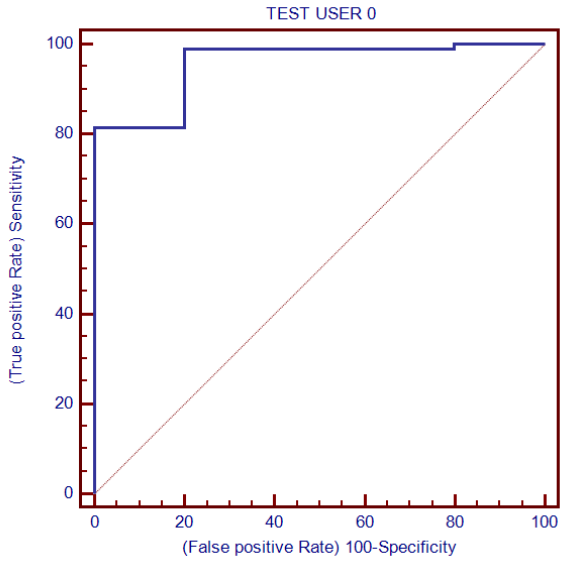


Figure 15: Receiver Operating Characteristic (ROC) Curve, User 0, $N = 2$

7.2 Using more data for testing

We experimented with the proportion of data used for training and testing. As we have discussed in earlier chapters, five fold validation uses four sets for training and just one set for testing. We thought of using more data for validation purpose and see if the results vary because of this change. We have used three fold cross validation for this, that is, we kept two sets aside for validation and we performed three fold validation on remaining three sets. So in total we had three sets for testing.

We observed that for most of the users, the graphs for five fold validation (with less testing data) and three fold validation (with more testing data) are almost identical. For some users, there is slight change in the result when more data is used for validation.

For some of the users five fold cross validation shows better results. Figure 12 shows five fold cross validation results for User 0 where we can see separation whereas,

Figure 16 shows results for three fold cross validation for same user where one point had been misclassified as a masquerader.

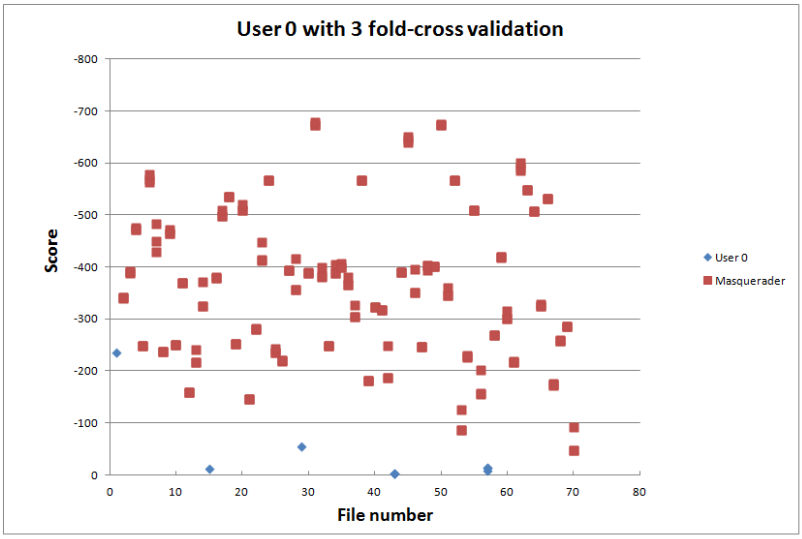


Figure 16: User 0, 3 fold cross validation (more validation data)

Let us look at some cases where three fold cross validation gives better results over five fold cross validation. Lets consider User 5 first. Figure 17 shows graph for five fold cross validation (with less validation data). As we can see due to the point which is closer to -100, we can not draw a line which separates User 5 (good user) and all other masqueraders.

Figure 18 shows the graph for same user, that is, User 5 but with more validation data (three fold cross validation). Here we can see that now there is a complete separation between the two classes, which is a good thing. This seems somewhat counterintuitive but we think that this could be because of changing user command history over time and since in three fold cross validation we used lesser history for training, the model better captured the user profile.

Similarly, Figure 19 and Figure 20 shows differences for User 9. Here for less data there is no complete separation whereas with more validation data, we are getting

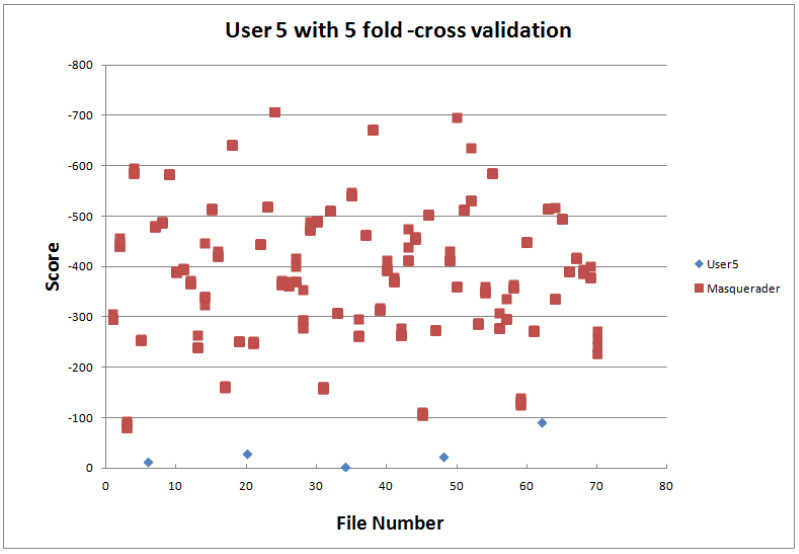


Figure 17: User 5, 5 fold cross validation (less validation data)

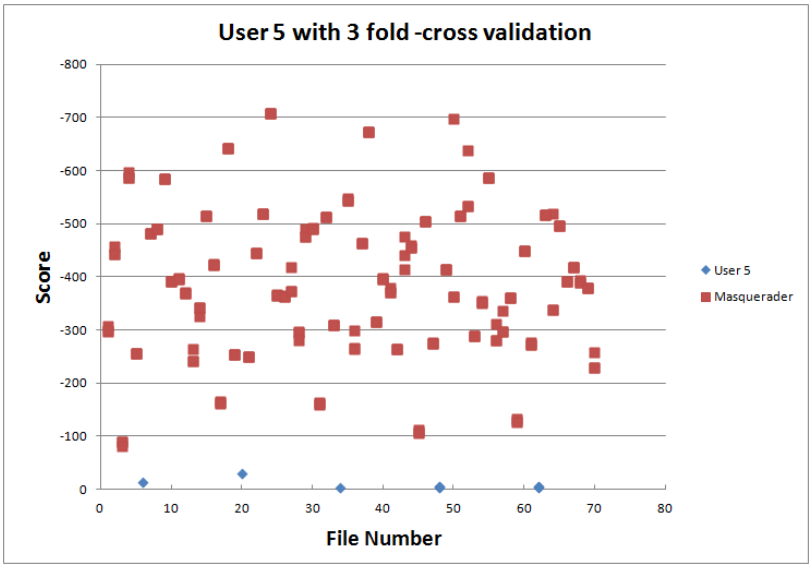


Figure 18: User 5, 3 fold cross validation (more validation data)

better results over the earlier case.

7.3 Effect of varying testing data sequence size

This section discusses the correlation between length of test sequence and its effectiveness. We have generated five different cases for variable length testing data

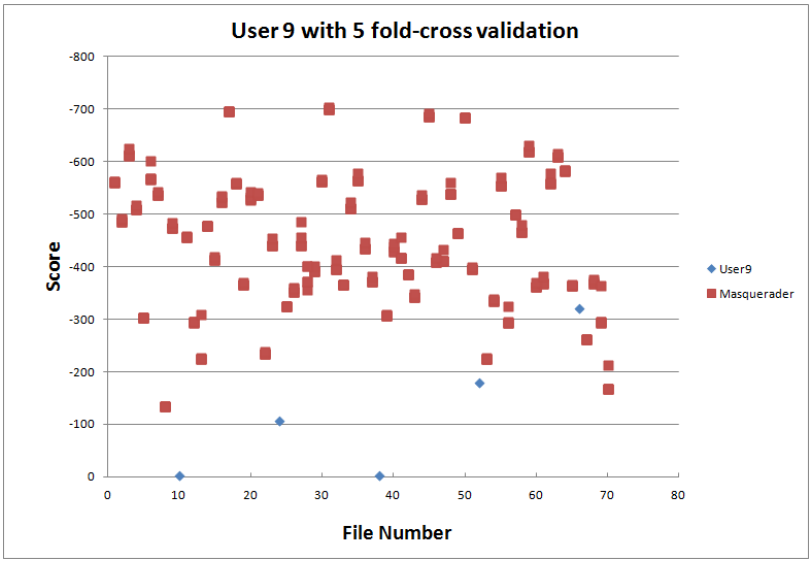


Figure 19: User 9, 5 fold cross validation (less validation data)

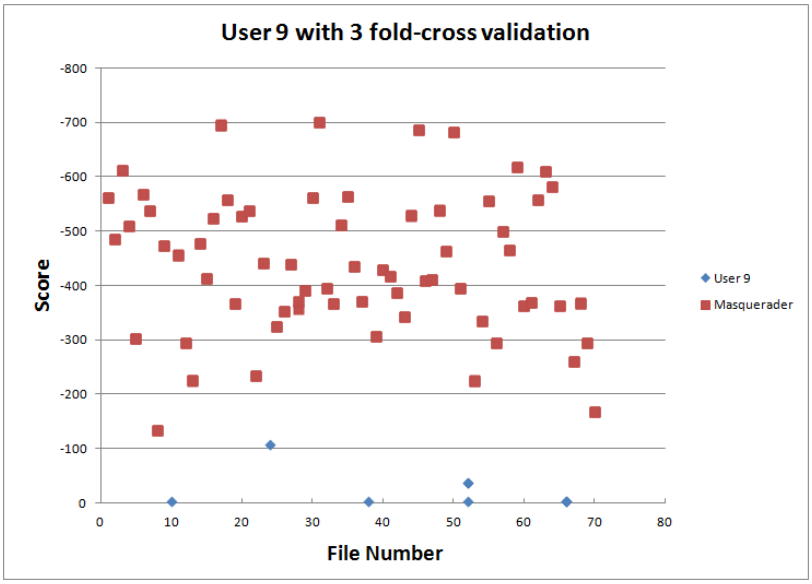


Figure 20: User 9, 3 fold cross validation (more validation data)

sequences. Table 3 refers to these five cases. In case 1, the testing data set is divided in two parts so third column is 50% .

Keeping the training data constant in size, we are reducing length of the scoring dataset to see if this changes our earlier results. Figure 21 shows the experimental

Table 3: Experiment Cases For Generating Multiple Test Sequences

Case Number	Number of Sequences	% Test Data per sequence
1	2	50 %
2	4	25 %
3	10	10 %
4	15	6.6 %
5	20	5 %

results for User 3. We reduced testing data length for User 3 from 1864 to 93 i.e. Case 5. As we can see here, more the testing data better the results given by HMM. Similarly, for less amount of testing data, the results are not that great.

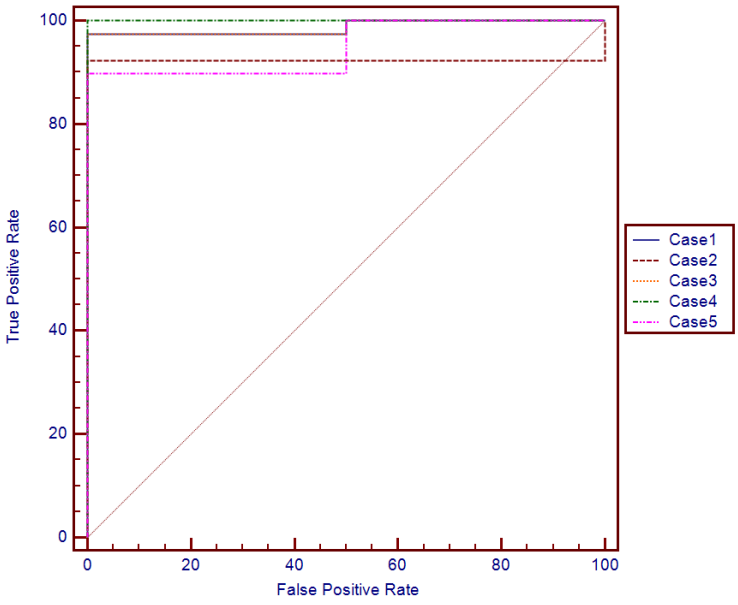


Figure 21: Comparison of different test length sequences, User 3

Figure 22 shows similar results for User 11.

We observed that, for 40% of users out of total, there is no significant effect of varying the testing data sequence. Figure 23 shows such an example. For User 13, there is no significant change in results when the testing sequence size is varied. This

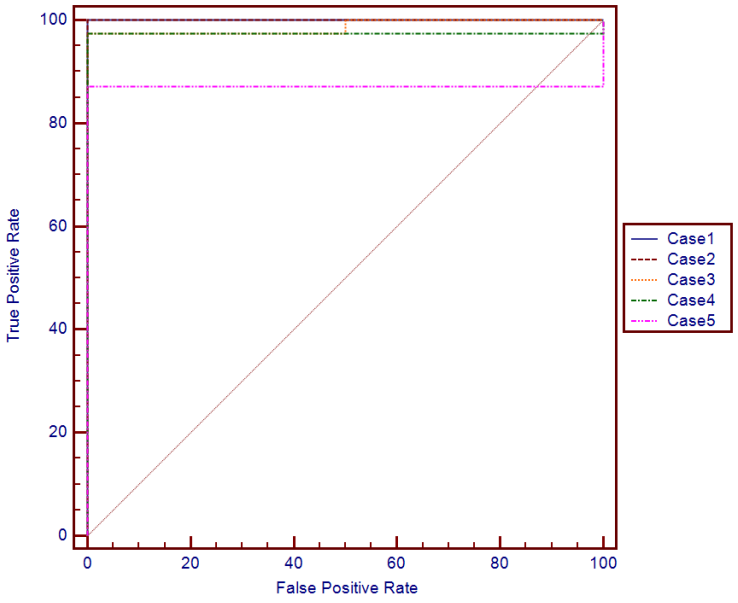


Figure 22: Comparison of different test length sequences, User 11

might be because this user has large number of commands as compared to other users. We were not able to include more cases for this experiment because of the limited number of UNIX commands for some users.

7.4 Comparison with Schonlau Dataset

We ran our experiments on Schonlau dataset to see if the results vary from the ones we got with our dataset. Figure 24 and Figure 25 shows the experimental results for Schonlau dataset's User 1 and User 7.

The results using Schonlau's dataset and using our new dataset are similar. For Schonlau dataset we observed good separation for fewer users but in our case, results for 50% of users have complete or good separation between good users and masqueraders. One of the reasons for lesser number of users getting good separation in case of Schonlau dataset could be that the dataset has more number of users along

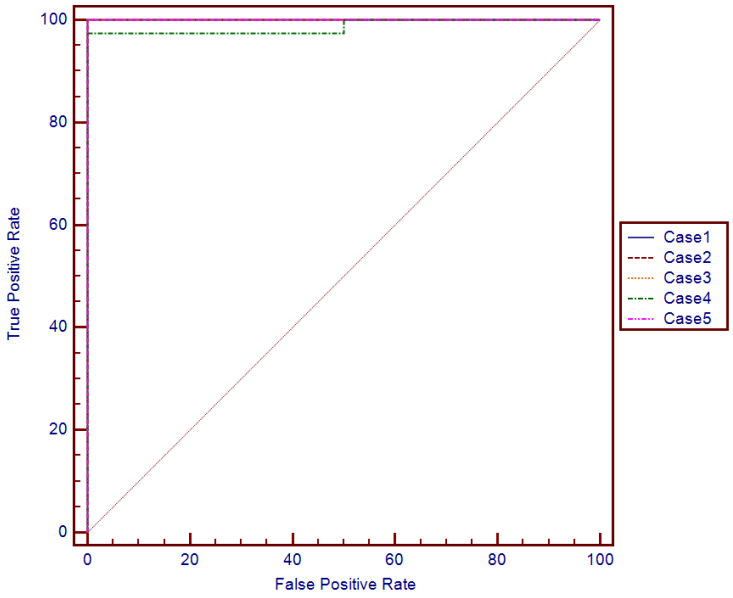


Figure 23: Comparison of different test lengths sequences, User 13

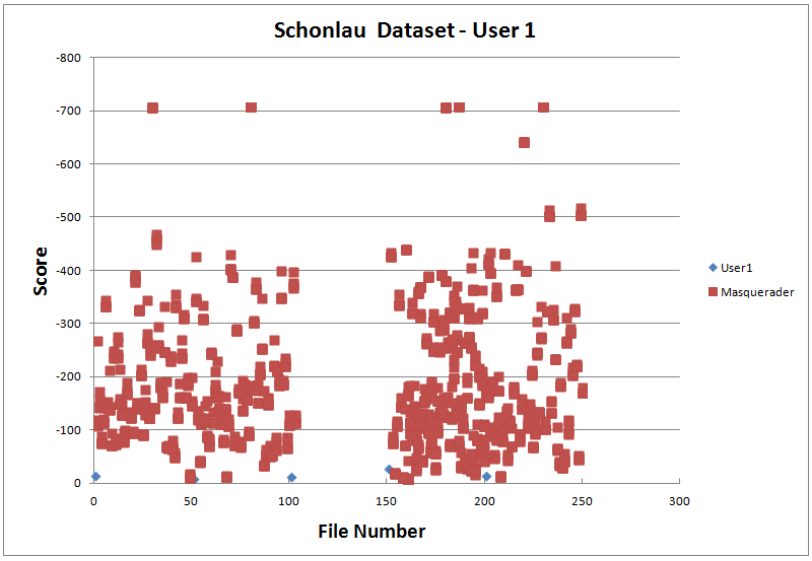


Figure 24: Experimental Results for Schonlau Dataset, User 1

with large number of commands per user. So there is higher probability of users using similar commands and thus having higher scores.

We also plotted ROC for one of the users' in Schonlau dataset. Figure 26 shows

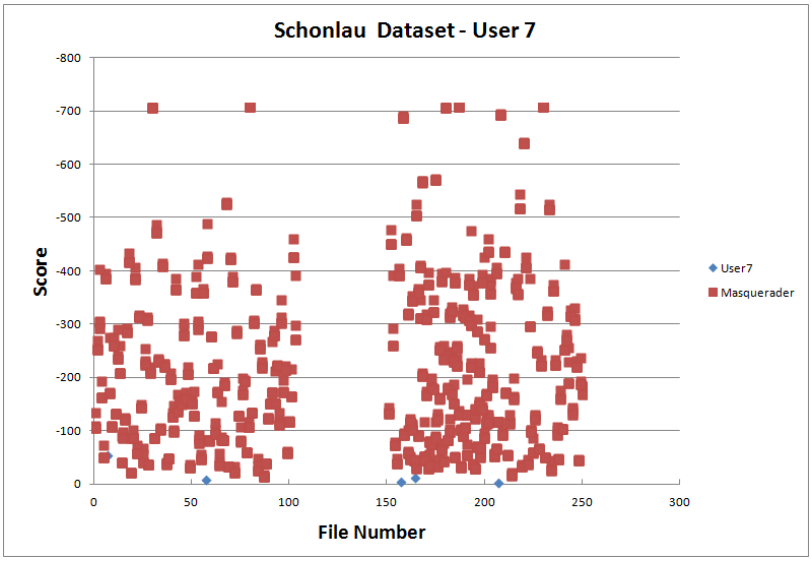


Figure 25: Experimental Results for Schonlau Dataset, User 7

the ROC. AUC for this particular user is 0.89.

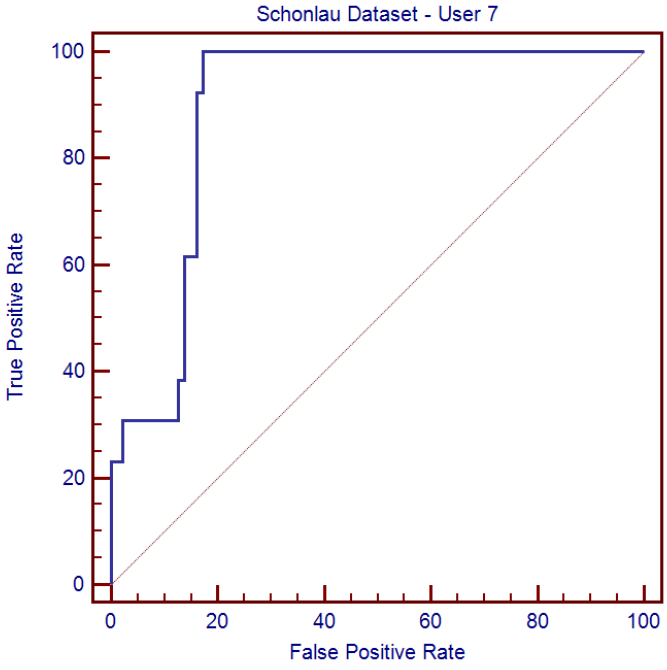


Figure 26: ROC for Schonlau Dataset, User 7

CHAPTER 8

Conclusion

In this project, we have created a new UNIX commands dataset which will only include UNIX commands issued by a user and not the commands executed by the system. This new dataset also has command execution timestamp and arguments in addition to the commands themselves. We used HMM for masquerader detection on this data and ran a number of tests on this new dataset to see if it is comparable to Schonlau dataset. We have analyzed the results and concluded that the results with this dataset are good and similar to that of the Schonlau dataset. For few users, the results are not that good. We think the reason for this might be lesser number of commands for those users. Another reason might be, these users executed commands that are similar to some other users. Due to which a conservative threshold calculation results in more false negatives.

We also ran experiments for different number of states of HMM and found that number of states does not have significant impact on the results. The HMM with two hidden states performs slightly better than other states for some users.

We experimented with size of both training and testing data set sequences. We conjectured that the results are better when more training data is used for most cases. Similarly, we observed that the results are better when longer test command sequences are used over shorter ones.

Initially we were thinking to use a dataset that had a similar structure to the Schonlau dataset. This would imply that, the testing data is mixed with other users (not in training dataset). When we analyzed our dataset we found that lot of users

have similar commands, so as a start we decided to consider other user's data as masquerader data instead of creating intrusion data by inserting other users commands with good users commands. One future is to use the timestamp and arguments in conjunction with the commands issued by users and test if the results improve. Secondly, create masquerader dataset by mixing good and other user's data and use that for testing purpose.

LIST OF REFERENCES

[1] Alpaydin, E., (2010). "Introduction to Machine Learning". Retrieved from http://www.realtechsupport.org/UB/MRIII/papers/MachineLearning/Alppaydin_MachineLearning_2010.pdf

[2] Ataman, K., Zhang, Y., (2006). "Learning to rank by maximizing AUC with linear programming", in International Joint Conference on Neural Networks. pp. 123-129.

[3] Axelsson, S., Jonsson, E., et al., "An Approach to UNIX Security Logging". Rtrieved from http://www.securiteinfo.com/ebooks/pdf/unix-sec-log.pdf

[4] Bertacchini, M., Fierens, P., (2009). "A Survey on Masquerader Detection Approaches". Retrieved from http://www.criptored.upm.es/cibsi/cibsi2009/docs/Papers/CIBSI-Dia2-Sesion5(2).pdf

[5] Cortes, C., Mohri, M., (2004). "AUC optimization vs. error rate minimization" in NIPS, S. Thrun, L. K. Saul, and B. Scholkopf, Eds., vol. 16. The MIT Press. pp. 313-320.

[6] Egan, J., (1975) "Signal Detection Theory and ROC Analysis". Academic Press, Inc.

[7] Erbacher, R., Shashi, P., Chet, C., (2009). "Intrusion Detection: Detecting Masquerade Attacks Using UNIX Command Lines". Retrieved from http://digital.cs.usu.edu/~erbacher/publications/MasqueradeDetectionConference.pdf

[8] Garfinkel, S., Spafford, G., (1996). "Practical Unix & Internet Security (2010)". Retrieved from http://docstore.mik.ua/orelly/networking/puis/ch10_02.htm?&lang=en_us&output=json&session-id=37917ac8f4dc8fb1f45be60484586c65

[9] Geng, D., Odaka, T., (Springer 2010). "An N-Gram and STF-IDF model for masquerade detection in a UNIX environment".

[10] Kemp, S., (2009). "BASH history forever. Debian Administration". Retrieved from http://www.debian-administration.org/articles/175

[11] Kothari A., (2012) "Defeating Masquerade Detection", Masters thesis, San Jose State University. Retrieved from http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1239&context=etd_projects

[12] Krogh, A., Brown, M., et al., "Hidden markov models in computational biology: applications to protein modeling", J. Mol. Biol., vol. 235, no. 5, pp. 1501-1531, 1994.

[13] Lee, W. and Stolfo, S. "Data mining approaches for intrusion detection". In Proc. 7th USENIX Security Symposium, 1998.

[14] Lin, H., Stamp, M., (November 2011). "Masquerade detection using profile hidden Markov models, Computers and Security", Vol. 30, Issue 8, pp. 732-747

[15] Natarajan , R., (2008, August). "15 Examples To Master Linux Command Line History". Retrieved from http://www.thegeekstuff.com/2008/08/15-examples-to-master-linux-command-line-history/

[16] Schonlau, M., (2001) "Masquerading User Data", Retrieved from http://www.schonlau.net/intrusion.html

[17] Stamp, M., (2012, February). "A Revealing Introduction to Hidden Markov Models". Retrieved from http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf

[18] Stamp, M., Lin, H., (2010, December). "A Study on Masquerade Detection". The Faculty of the Department of Computer Science San Jose State University. Retrieved from http://www.cs.sjsu.edu/faculty/stamp/students/huang_lin.pdf

[19] The GNU Accounting Utilities, (2010). "Free Software Foundation, Inc.", Retrieved from http://www.gnu.org/software/acct/

[20] Toderici, A., (2012). "Chi-squered distance and metamorphic virus detection", Masters thesis, San Jose State University. Retrieved from http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=7710&context=etd_theses

[21] Ubuntu Server Guide (2006). "Canonical Ltd". Retrieved from https://help.ubuntu.com/8.04/serverguide/backup-shellscripts.html

[22] Ubuntu Documentation. Last visited (2012, June 5). Retrieved from https://help.ubuntu.com/community/CronHowto

[23] Wong, W., (2006). "Analysis and Detection of Metamorphic Computer Viruses", Masters thesis, San Jose State University. Retrieved from http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf

APPENDIX A

Scripts Used To Collect Data

A.1 Setup

The following script will install `pacct` package on user's machine and will also do necessary settings for `.bash_history` file.

numbersep

```
#!/bin/bash
#part1 - pacct package setup
USER_HOME=$(eval echo ~${SUDO_USER})
BASHRC_LOCATION=$USER_HOME/.bashrc
DEFAULT_ACCT_FILE=/etc/default/acct
BASEDIR=$USER_HOME/amruta_masquerade_detection_project

if [ -f /etc/debian_version ];
then
    DISTRO=Debian
    echo ``Installing acct for $DISTRO''
    sudo apt-get install acct
else
    echo ``This operating system is not supported''
    exit -1
fi

#change ACCT_LOGGING=``30'' to 60
sudo sed -i `s/ACCT_LOGGING=\``.*\''/ACCT_LOGGING=\``60\''/'
```

```
    $DEFAULT_ACCT_FILE


#*****************************************************
#CHANGE THE BASH_HISTORY SO THAT IT DISPLAYS TIMESTAMP
#*****************************************************
echo ``export HISTTIMEFORMAT = ' %F %T ' '' >>$BASHRC_LOCATION
#*****************************************************
#CHANGE THE BASH_HISTORY SO THAT BASH HISTORY FILE SIZE IS
    SUFFICIENTLY LARGE
#*****************************************************
echo ``export HISTFILESIZE=20000'' >>$BASHRC_LOCATION
echo ``export HISTSIZE=20000'' >>$BASHRC_LOCATION
#part2 - cronjob setup (execute cronjob -e on  user's machine)
(crontab -l ; echo ``00 12 * * * $BASEDIR/scripts/
    copy_history_files.sh >> $BASEDIR/scripts/copy_history_files.
    log 2>&1'') | crontab -
```

setup.sh

A.2 Copy History Files

The following script will collect .bash_history.

numbersep

```
#!/bin/bash
#Copy /home/$USER/.bash_history to desired location
#Auto Archive countdown move loop
#This script will create $count number of backups of a given file
count=120
move=$[ $count - 1 ]
USER_HOME=$(eval echo ~${SUDO_USER})
BASEDIR=$USER_HOME/amruta_masquerade_detection_project
sfolder=''$USER_HOME''
dfolder=''$BASEDIR/history_files''
file=''.bash_history''
date

#check if the backup folder is present
if [ ! -d $dfolder ]
then
        mkdir $dfolder
fi

# Automate until $count = 0
while [ $count -gt 0 ]
do
  if [ -e $dfolder/$file.$move ]
```

```
24    then
        echo ``Moving $dfolder/$file.$move to $dfolder/$file.$count''
26      mv $dfolder/$file.$move $dfolder/$file.$count
      fi
28    count=$[ $count - 1 ]
      move=$[ $move - 1 ]
30  done
  echo ``Copying $sfolder/$file to $dfolder/$file.0''
32  cp $sfolder/$file $dfolder/$file.0
  #EOF
```

copy_history_files.sh