ABSTRACT

Title of Thesis: DETECTION AND CLASSIFICATION OF NETWORK INTRUSIONS USING HIDDEN MARKOV MODELS

Degree candidate: Svetlana Radosavac

Degree and year: Master of Science, 2002

Thesis directed by: Professor John S. Baras
Department of Electrical Engineering

With the increased use of networked computers for critical systems, network security is attracting increasing attention and computer network intrusions have become a significant threat to communication and computer networks in recent years.

The models developed in this thesis represent the first step in modelling of network attacks. The thesis demonstrates that models that represent network attacks can be developed and used for both detection and classification. In this thesis we put emphasis on detection and classification of network intrusions and attacks using Hidden Markov Models and training on anomalous sequences. We

test several algorithms, apply different rules for classification and evaluate the relative performance of these. We put emphasis on one particular classification algorithm that is not dependent on data set properties. Several of the attack examples presented exploit buffer overflow vulnerabilities, due to availability of data for such attacks. We demonstrate that models for other attacks can be built following our methods but could not be tested due to lack of data.

The new method proposed in this thesis is highly efficient and captures characteristic features of attacks in short period of time using very low number of sequences.

DETECTION AND CLASSIFICATION OF NETWORK INTRUSIONS USING HIDDEN MARKOV MODELS

by

Svetlana Radosavac

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2002

Advisory Committee:

Professor John S. Baras , Chairman/Advisor
Professor Virgil D. Gligor
Professor Carlos A. Berenstein

Dedication

To my parents, for their everlasting love and support

Acknowledgements

I would like to express my honest thanks to my advisor, Professor John Baras for his guidance and valuable comments throughout this work. I would also like to thank Dr Carlos Berenstein and Dr Virgil Gligor for agreeing to serve in my committee. I am also grateful to Maben Rabi, Alvaro Cardenas, Trevor Vaughan and all other members of the Distributed Immune Systems group for their help and valuable comments. Many thanks to Iordanis Koutsopoulos for his friendship and support throughout my studies in Maryland. Without his support, I probably wouldn't be where I am now.

I am grateful for the support of my research work and graduate studies by the CIP-URI DoD ARO grant DAAD19-01-10494 through the University of Maryland College Park

Table of Contents

List of Tables

List of Figures

DETECTION AND CLASSIFICATION OF NETWORK INTRUSIONS USING HIDDEN MARKOV MODELS

Svetlana Radosavac

December 9, 2002

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` class by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

In this chapter we discuss the importance of computer network security and present the basic types of intrusion detection systems and their main features. We present an attack taxonomy and explain the terminology used in this thesis. Finally we present the motivation for the thesis and outline the remaining chapters.

1.1 Computer Network Security

With the increased use of networked computers for critical systems, computer network security is attracting increasing attention and network intrusions have become a significant threat in recent years. Computer network security is primarily concerned with protecting a particular resource: valuable data or valuable information. The number of intrusions is dramatically increasing and so are the costs associated with them. The number of incidents reported to Carnegie Melon's Computer Emergency Response Team/Coordination Center (CERT/CC) has increased from the range of 2000-3000 in early and mid 1990s to 9859 in 1999, 21,756 in 2000, 52,658 in 2001 and the data for 2002 indicate the number of incidents is in the range of mid 70000s. With an increased understanding of how systems work intruders have become more skilled at determining weaknesses in systems and exploiting them to obtain increased privileges. It is important to mention that the knowledge necessary to carry out an attack is decreasing, resulting in a rapid increase of attempted attacks on various systems and networks. On the other hand, the increasing frequency and complexity of Internet attacks has raised the level of knowledge required by systems and network administrators to effectively detect intrusions. Another problem that has appeared in the last couple of years is the appearance of multi-stage attacks that can be orchestrated to strike multiple targets with different levels of security by coordinating a number of exploits. An additional difficulty in detecting attacks is due to the fact that the majority of intruders attacking high profile targets conceal the true origin of an attack and they rarely indulge in sudden bursts of suspicious activity that can be easily detected, even by very simple intrusion detection systems.

The ideal approach to securing the network is to remove all security flaws from individual hosts, but less expensive and more feasible approaches are used for securing the computer systems. Most computer systems have some kind of security flaw that may allow intruders or legitimate users to gain unauthorized access to sensitive information. Moreover, many computer systems have widely known security flaws that are not fixed due to cost and some other limitations. Even a supposedly secure system or network can still be vulnerable to insiders misusing their privileges or it can be compromised by improper operating practices.

Intrusion detection is based on the fact that an intruder's behavior will be significantly different from that of a legitimate user. A more precise definition of computer security is based on the realization of confidentiality, integrity and availability in a computer system. Confidentiality requires that access to information is restricted to users authorized for it (encryption, authentication and authorization). Integrity requires the stored information not to be altered and availability requires the computer systems to be available when the authorized users need them.

The primary sources of information for intrusion detection systems are network activity and system activity. Network-based systems look for specific patterns in network traffic and host-based systems look for those patterns in log files. In general, network-based IDS can detect attacks that host-based systems can miss because they examine packet headers and the content of the payload, looking for commands or syntax used in specific attacks. Another advantage of network-based IDS is that it is easy to manipulate the log files in host-based systems and cover traces of the attack, but it is not possible to do so in network-based systems. Also, network-based systems detect rejected attacks, whereas host-based IDS are not able to do that. On the other hand, host-based systems can verify success or failure of the attack, detect specific system activities that a network-based system is not able to detect and are able to detect encrypted attacks. This leads us to the conclusion that the best IDS is the mixture of host and network-based systems.

We can group intrusion detection systems into two classes: *anomaly detection systems* and *signature based systems*. Anomaly detection systems attempt to model the usual or acceptable behavior. They have high false positive rates and usually have detection delay. Irregular behavior is flagged as potentially intrusive. Misuse detection refers to intrusions that follow well-defined patterns of attack that exploit weaknesses in the system. Misuse detection techniques have high detection rate for already known attacks but even the slightest variation of an attack signature may cause the system not to recognize the attack. More detailed description of anomaly and misuse detection systems will be presented in the following chapter.

1.2 Attack planning and execution

To be able to mount a successful attack on the remote system an attacker has to collect maximum amount of information about the target system and gain remote or local access to the system. In order to gain information about Internet (domain name, IP addresses of systems reachable via Internet, TCP and UDP services running, system architecture, system enumeration), intranet (networking protocols used etc.), remote access (phone numbers, authentication mechanisms etc.) and extranet (type of connection, access control mechanism), the attacker has to perform footprinting of the target system [1]. Footprinting has several stages. The initial stage after determining the scope of activities (whether he will scan the whole organization, a subdomain of organization etc.) is network enumeration, where the attacker identifies domain names and networks related to the system he is footprinting. After successfully performing the network enumeration and identifying all the associated domains he performs DNS interrogation and finally network reconnaissance which allows the attacker to determine network topology of the discovered networks and potential access paths into the network. Once the attacker has performed network footprinting, scanning and enumeration, he at least has the information about the system that is running and most likely more detailed information about programs installed and therefore by looking at the list of vulnerabilities of the system he can attack the network or particular hosts. Less skillful attackers will try to exploit already discovered vulnerabilities hoping that the systems have not yet been patched. That would result in either unsuccessful attacks due to the fact that, for example, the attacker mounted an attack written for UNIX system to a Windows machine, or in the successful exploit due to the fact that the attacker found a network that doesn't have a patched version of the system/software. This type of attacks can be prevented by following newly released lists of vulnerabilities and applying patches on the system. However, more skillful attackers will perform the full scanning of the system, find out which system is used, find out all potential vulnerabilities and apply their own, newly created attacks. A special threat for the security of UNIX computer systems represent stealthy User-to-Root Attacks, which were included in the 1999 DARPA evaluation data set [4]. The main purpose of stealthy attacks is to avoid creating familiar attack patterns that make the attacks easy to detect. In order to avoid detection stealthy attacks should only open and modify files that are accessed frequently, execute only commonly used programs, not create new shells owned by root, not execute unusual system calls (or sequences of system calls) or not create unusual traffic. Network traffic created by an attack can be monitored using tcpdump. According to [2] the most important guidelines for creating stealthy User-to-Root attacks are:

1. Attack scripts should be camouflaged during transfer using simple forms of encryption;

2. All attack commands should be encapsulated into one shell script;

3. The output of an attack should not be displayed;
4. Mail, HTTP and other common services should be used for data transfer;
5. Encrypted attack exploits should be created using common editors or shell scripts;
6. Attacks should be spread over multiple sessions and time;
7. Attacks should use common commands and network services;
8. The stealthiness of each attack should be confirmed;
9. Probes should be executed slowly, from multiple sources in random order.

In most cases it is possible to detect all or most of known attacks and construct systems that will successfully detect them, but it is a challenging task to construct a system that would be able to detect new attacks.

1.3 Attack taxonomy

Kristopher Kendall's thesis [3], based on the 1998 DARPA intrusion detection evaluation set has a detailed overview of attacks and their classification. With the taxonomy presented in [3] each attack can be classified as one of the following:

- Action at one level of privilege;
- Unauthorized transition (from a lower privilege level to a higher privilege level);
- Action at a higher level of privilege, while the user is at the lower privilege level.

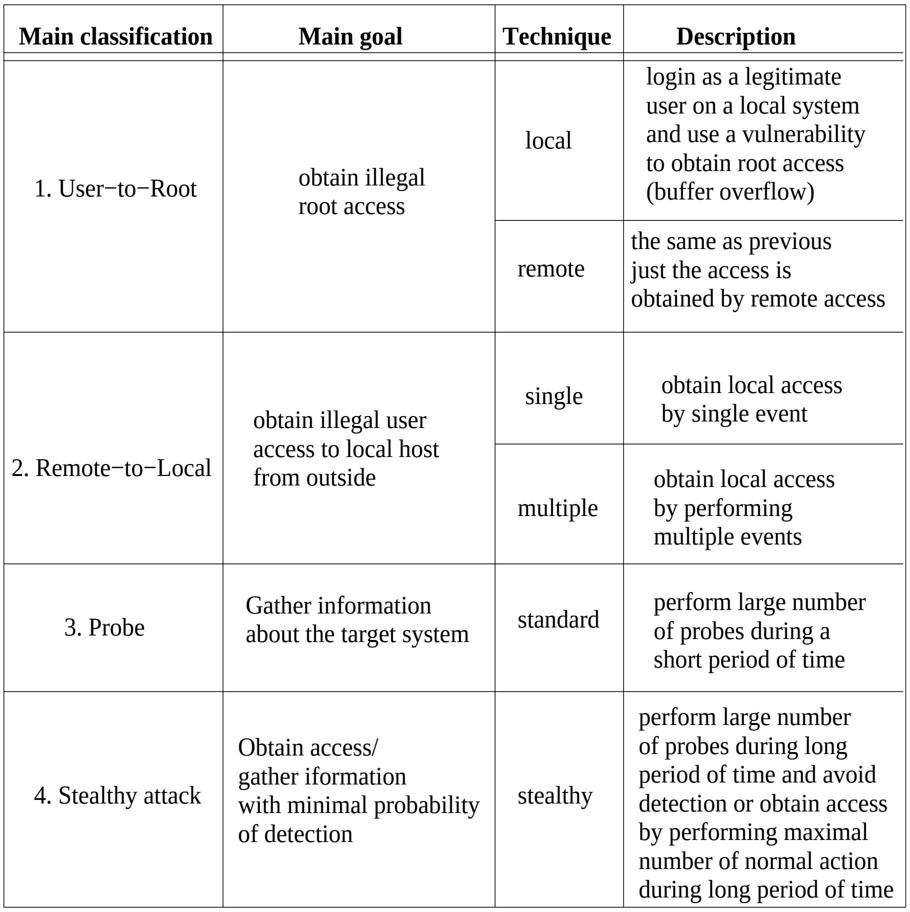
In order to perform an attack the attacker needs to gain access to the local host and exploit a vulnerability in the system. Some methods of transition and exploitation that were used in the 1998 DARPA evaluation are:

- **m - Masquerading**: an attacker gains access to a system by misrepresenting himself;
- **a - Abuse of Feature**: a user performs extremely high number of legitimate actions that can lead to system congestion or program failure;
- **b - Implementation Bug**: bugs in programs can lead to compromising the whole system. Typical examples are buffer overflows and race conditions;
- **c - System Misconfiguration**: access is gained due to system misconfiguration (i.e. when some accounts are left with *guest*, *anonymous* or no password at all);
- **s - Social Engineering**: access is gained by fooling an authorized user into providing information that can be used to break into the system.

In summary, using the taxonomy presented in [4] attacks can be classified as shown on figure 1.1.

1.4 MIT Lincoln Labs Evaluations

In the MIT Lincoln Labs report [4] results of eight different IDS systems ([5, 7, 8, 9, 10, 11, 12]) were compared. Systems [5, 8, 12] used BSM Solaris host audit data to detect Solaris R2L (Remote-to-Local) and U2R (User-to-Root) attacks.



Main classification	Main goal	Technique	Description
1. User−to−Root	obtain illegal root access	local	login as a legitimate user on a local system and use a vulnerability to obtain root access (buffer overflow)
		remote	the same as previous just the access is obtained by remote access
2. Remote−to−Local	obtain illegal user access to local host from outside	single	obtain local access by single event
		multiple	obtain local access by performing multiple events
3. Probe	Gather information about the target system	standard	perform large number of probes during a short period of time
4. Stealthy attack	Obtain access/ gather iformation with minimal probability of detection	stealthy	perform large number of probes during long period of time and avoid detection or obtain access by performing maximal number of normal action during long period of time

Figure 1.1: MIT Attack Taxonomy.

Systems [8, 12] produced a combined output from both network sniffer data and host audit data and used network sniffer data to detect R2L and U2R attacks against the UNIX victims. System [9] used NT audit data to detect U2R and R2L attacks against the Windows NT victim and [8, 12] used BSM audit data to detect Data attacks against the Solaris victim. System [11] used information from a nightly file system scan to detect R2L, U2R and Data attacks against the Solaris Victim. Three weeks of training data were used, composed of two weeks of background traffic with no attacks and one week of background traffic with a few attacks. The results showed that many stealthy and new attacks were frequently missed. When designing stealthy attacks, they tried to avoid obvious patterns that make those attacks easy to detect. It was shown that detection rates for U2R and Data attacks are generally low for SunOS and Linux victims where extensive audit data is not available. They also compared performance of the best IDS in each category (DoS, Probe, R2L and U2R) for old and new and for old and stealthy attacks. The detection of new attacks was much worse than detection of old attacks, especially for DoS, R2L and U2R attacks. The average detection rate for old attacks was 72% and for new attacks only 19%. Also, stealthy probes and U2R attacks were much more difficult to detect for network-based IDS that used sniffing data. U2R attacks against a Solaris victim were accurately detected by host-based intrusion detection systems that used BSM audit data. Attacks were not detected for various reasons. Signature-based systems missed new attacks due to inability to recognize variations of old attacks or because the system did not have a signature for the new attacks. Stealthy probes were missed because they are executed during extended periods of time and the systems were trained to recognize only aggressive probes that exceeded certain thresholds.

1.5 Terminology

The field of intrusion detection is relatively young and many common terms have a number of meanings. Therefore, this section presents some basic definitions used in this thesis to avoid ambiguity if multiple definitions of one term exist.

Audit

(1) to examine a system for security problems and vulnerabilities. (2) to record and analyze system activity for security problems and vulnerabilities [6]. The first defined audit activity is referred to as static audit. The second definition refers to the dynamic activity of monitoring the system state as it changes over time. Throughout the remainder of this thesis the second definition will be used.

Audit trail

A chronological set of records of system activity [6]. In this thesis audit trail is a record of system activity.

Buffer overflow

Redirection of the flow of execution of a program to perform some chosen activity by feeding the program with selected input values.

False negative

Intrusion that is not detected by the intrusion detection system and is classified as normal activity.

False positive (False alarm)

Normal activity that is classified as intrusion.

Intrusion

(1) inappropriate use of a computer system. (2) penetration of a computer system by an outsider [6]. Throughout this thesis the second definition will be used.

Intrusion detection

(1) identifying individuals who are using or attempting to use the computer system without authorization or who have legitimate access but are attempting to abuse their privileges.

Misuse

Inappropriate use of the computer system.

Misuse detection

Identification of any attempted improper or inappropriate use of the system.

Normal

There are many definitions of normal. In this thesis normal behavior will be the behavior that is not intrusive (does not match the training set of intrusive behavior).

1.6 Motivation

There exists a sustained need for attack detection and in cases when the attacks do happen we need to classify them. This thesis demonstrates that it is possible to model attacks with a low number of states and classify them using Hidden Markov Models with very low False Alarm rate and very few False Negatives. Current results through our methods and algorithms do not display any False Negatives, but we cannot claim that False Negatives will not appear in future applications. The attacks for which we had most instances and associated data were attacks exploiting buffer overflows and that is the reason why our experimental results and applications of our algorithms presented in this thesis are for such attacks. We present performance and experimental results for buffer overflow attacks, while we describe how it is possible to apply our models and algorithms for detection and classification developed in this thesis to other attacks as well. We also emphasize that the purpose of our algorithms is not only the detection and classification of buffer overflows, but they are targeted for detecting and classifying a broad range of attacks (like Remote to Local attacks, race condition attacks, etc.). We are aware of the existence of tools for static code checking and their advantages over Intrusion Detection Systems [43, 44, 42]. If implemented correctly and in all programs all buffer overflows would be detected. However, they also have some disadvantages (high false alarms, high false negatives, cannot detect all network attacks etc.) and are not applied in all programs. A detailed analysis and comparison is presented in Chapter 3.

1.7 Thesis organization

The first chapter of this thesis provided background on current issues in the field of intrusion detection, attack taxonomy and attack organization. It also provided basic definitions that are relevant for this thesis. Chapter 2 discusses previous work in the field of intrusion detection systems and points out their strengths and weaknesses. Chapter 3 presents attack representation techniques and description of U2R and R2U attacks. Chapter 4 presents a short overview of the theory of Hidden Markov Models and multihypothesis testing. It presents the algorithm used for detection and classification of intrusions and its application for detecting buffer overflow attacks. Chapter 5 presents results of applying our algorithm to monitoring data and Chapter 6 presents the conclusions, contributions and possible extensions of this work.

Chapter 2

Literature overview

This section presents some background in the area of intrusion detection. The first section analyzes anomaly detection systems and their performance. It gives a short overview of statistical methods for anomaly detection, predictive pattern generation for anomaly detection and program-based anomaly detection. The second section presents an overview of misuse detection techniques: language-based misuse detection, expert systems and high-level state machines for misuse detection. The third section presents an overview of specification-based approaches to intrusion detection: process, behavior monitoring and process state analysis. The last section presents the "attack tree" approach of modeling attacks.

Each of the subsections gives a short overview of the technique of interest and then presents the data set used. At the end of each section a table comparing the main features of all methods is presented. The features presented in these tables are: approach, database of normal present/not present, profiles, rules, events, signatures, data set and type of classification.

This overview puts more weight to the approach presented in this thesis since it is the first attempt to detect the abnormality and classify it as a specific attack (ffbconfig, eject, fdformat or ps).

2.1 Anomaly Detection

2.1.1 Statistical Methods for Anomaly Detection

In statistical methods for anomaly detection the system observes the activity of subjects and generates profiles to represent their behavior, where a profile includes such measures as activity intensity measures, audit record distribution measures, categorical measures (the distribution of an activity over categories) and ordinal measure (such as CPU usage). Two profiles are maintained for each subject: the current profile and the stored profile. As the records are processed, the system updates the current profile and periodically calculates a measure of the activity's abnormality by comparing the current with the stored profile using a function of abnormality of all measures within the profile. The stored profile is periodically updated by merging it with the current profile.

Disadvantages of statistical methods are: some methods can be trained by a skilled attacker to accept abnormal behavior as normal, it can be difficult to determine thresholds that balance the likelihood of false positives with the likelihood of missing intrusive events, need accurate statistical distributions. Not all behaviors can be modeled using purely statistical methods, most techniques require the assumption of a quasi-stationary process, which is not always the case. Models that do not make this assumption are more complex and time-consuming. (Kumar 95)

Haystack

Haystack [13] is an example of a statistical anomaly-based IDS. It uses both user and group-based anomaly detection. It models system parameters as independent, Gaussian random variables. Parameters are considered as abnormal when they fall out of 90% of the data range for the variable. It maintains a database of user groups and individual profiles. If a user has not previously been detected, a new user profile with minimal capabilities is created using restrictions based on the user's group membership. It is designed to detect six types of intrusions: attempted break-ins by unauthorized users, masquerade attacks, penetration of the security control system, leakage, DoS attacks and malicious use.

Bayesian classification

Bayesian classification automatically determines the most probable number of classes for the data, continuous and discrete measures can be freely mixed. On the other hand, this approach does not take into account the interdependence of the measures, it is unclear whether the algorithm can perform classification incrementally as needed for on-line processing, suffers from difficulty to determine thresholds, susceptibility to be trained by a skillful attacker etc.

2.1.2 Predictive Pattern Generation for Anomaly Detection

Predictive pattern generation for anomaly detection takes into account the relationship and ordering between events. It assumes that events are not random, but follow a distinguishable pattern. Such methods learn to predict future activities based on the most recent sequence of events and flag events that deviate significantly from the predicted.

Machine learning

T. Lane and C. Brodley applied the machine learning approach to anomaly detection in [14]. They built user profiles based on input sequences and compared current input sequences to stored profiles using a similarity measure. To form a user profile, the approach learns characteristic sequences of actions generated by users. For learning patterns of actions the system uses the sequence as the unit of comparison. A sequence is defined as an ordered, fixed-length set of temporally adjacent actions, where actions consist of UNIX shell commands and their arguments. To characterize user behavior they used only positive examples. For the purpose of classification, sequences of new actions are classified as consistent or inconsistent in reference to sequence history using a similarity measure. To classify the attack, they classify the similarity of each input sequence, yielding a stream of similarity measures. If the mean value of the current window is greater than the threshold then it is classified as normal.

DATA SET

The data used in the experiments consisted of a set of UNIX shell command histories from four members of the Purdue lab in a four month period. They compared the specific user to other users and calculated the similarity measure.

Time-based inductive generalization

Teng, Chen and Lu [15] developed a technique that applies a time-based inductive learning approach to security audit trail analysis. The system developed sequential rules. The technique uses time-based inductive engine to generate rule-based patterns that characterize the normal behavior of users. Each event is one single entry in the audit trail and is described in terms of a number of attributes (event type, image name, object name, object type, privileges used, status of execution, process ID, etc). The events in the audit trail are considered to be sequentially related. Rules identify common sequences of events and the probability of each sequence being followed by other events. Rules with a high level of accuracy and high confidence are kept, less useful rules are dropped. They define two types of anomaly detection: deviation detection and detection of unrecognized activities. Low probability events are also flagged, using a threshold. Anomaly is detected if either deviation or unrecognized activity are detected.

This method can identify the relevant security items as suspicious rather than the entire login session and it can detect some anomalies that are difficult to detect with traditional approaches. It's highly adaptive to changes, can detect users who try to train the system, activities can be detected and reported within seconds. On the other hand, some unrecognized patterns might not be flagged since the initial, predicting portion of the rule may be matched.

2.1.3 Program-based Anomaly Detection

Program modeling

The program modeling approach (Garth Barbour Ph.D. thesis [16]) presents an algorithm that efficiently learns the program behavior. The author claims that the number of false positives never increases with additional training. If there is a false positive, the representation can learn to accept the run in the future without negatively impacting the performance of other runs. He also claims that his technique can detect novel attacks as they occur, without manual specification of program behavior or attack signatures. His algorithm learns an approximation of the program's behavior. The number of false positives decreases with training set size. He uses a non-statistical method since statistical methods would miss minor deviations.

Barbour presents an algorithm that learns program behavior, without any input parameters presented in addition to his algorithm, which would add more reliability to his results. Input parameters may add additional reliability to detection because in programs with significant number of system calls, we may have different arguments associated with those system calls. If a program with limited capabilities is observed, it does not generate too many different system calls and, hence it is easy to detect any misuse of that program. On the other hand, if we are dealing with a powerful program, it generates a significant number of system calls and therefore, it is more difficult to detect misuse of that program.

DATA SETS

He initially claimed that his algorithm outperforms other algorithms on DARPA/LL, UNM and his own artificially generated set. However, the final results do not show any comparison with the UNM data and DARPA/LL data set shows that the work of Ghosh [26] outperformed the Program Modeling approach.

Computational Immunology

Stephanie Forrest is investigating anomaly detection on system processes from the perspective of immunology [17, 18, 19]. In [17] the authors base their approach on the immunological mechanism of distinguishing self from non-self and use lookahead pairs. They take short sequences of system calls, called n-grams, that represent samples of normal runs of programs and compare them to sequences of system calls made by a test program. If any run has a number of mismatches that is higher than a certain number (percent), it is flagged as anomalous. They extended their work to variable length sequences, based on random generation of examples of invalid network traffic to detect anomalous behavior. Unlike Barbour [16], where there is no delay in detecting anomalies, the approach of Forrest has a certain delay due to the fact that the program computes the percent mismatch to be able to flag or not flag the run as anomalous. Hofmeyr [18] used the rule of r contiguous bits and showed that fixed length sequences give better discrimination than lookahead pairs.

Christina Warrender [19] modelled normal behavior of data sets described below using stide, t-stide, RIPPER and HMMs. She compared learning methods for application to n-grams: enumerating sequences using lookahead pairs and contiguous sequences, methods based on relative frequency of different sequences, RIPPER (a rule-learning system) and HMMs. They used RIPPER to learn rules representing the set of n-grams. Training samples consisted of set of attributes describing the object to be classified and a target class to which the object belongs. RIPPER takes training samples and forms a list of rules to describe normal sequences. For each rule a violation score is established. Each set that violates a high confidence rule is a mismatch. They created HMMs so that there is one state for each n-gram. The results showed that HMMs performed marginally better than other methods. She showed that HMMs performed only marginally better than other simple and not computationally expensive methods and concluded that the choice of data stream (short sequences of system calls) was more important than the particular method of analysis.

DATA SETS

In [17] and [18] synthetic data sets were used. They were generated in production environment by running a prepared script. Warrender [19] collected traces of programs collected during normal usage and used only those programs that run with privilege due to the fact that misuse of privileged programs does the most harm to the system. She used `lpr, named, xlock, login` and `ps`. Data for `lpr` was collected at two universities using different printing environments. The `named` normal data consists of a single daemon trace and traces of its subprocesses and the intrusion against `named` is buffer overflow. They used two sample traces of buffer overflow attack. Data for `xlock` includes 71 synthetic traces and one live trace. The intrusion used was a buffer overflow. For `login` and `ps` they used two Trojan programs.

Variable-length audit trail patterns

Wespi [23] points out that usage of fixed-length patterns to represent the process model in [17] has no rule for selecting the optimal pattern length. On the other hand, Forrest *et.al.* [18, 19] point out that the pattern length has an influence on the detection capabilities of the IDS. Wespi *et.al.* initially were not able to prove significant advantage of variable-length patterns in [21]. Later, in [23] they showed that the previously stated method outperforms the method of Forrest *et. al*[18, 19]. They use functionality verification test (FVT) suites provided by application developers to generate a training set based on the normal program's specified behaviors. Their system consists of an off-line and on-line part. In the next step they apply the Teiresias algorithm [22] to generate the list of maximal variable-length patterns followed by some pruning during their preprocessing. During operation, a recursive pattern matching algorithm attempts to find the best matches for the test run. The measure of anomaly is the longest sequence of unmatched events (more than six consecutive unmatched events are treated as an intrusion).

Initial work generated variable-length patterns using a suffix-tree constructor, a pattern pruner, and a pattern selector. ID based on this technique can detect the attacks but is prone to issue false alarms. Then they used a modified version of the pattern generator to create variable-length patterns that reduce the number of false alarms substantially without reducing the ability to detect intrusions.

The technique presented in the initial paper had the weakness of not having a model for intrusion. Because of that reason, the advantages of the Teiresias algorithm were of no use and the performance was only marginally better than the methods of Stephanie Forrest.

The authors claim that their algorithm performs better than the fixed-length sequences of Stephanie Forrest, but they compare their algorithm only for the `ftpd` process and then claim that their algorithm outperforms algorithms presented in [17, 18]. This is arguable because it might be the case that they presented one example where their algorithm outperformed the other algorithms and did not present the instances where their algorithm failed. Hence, we can conclude that on that specific process algorithm of Wespi *et. al.* [23] did outperform the algorithm of Forrest *et. al.* [17, 18] but, on the other hand, that may or may not be true for other processes.

DATA SET

Wespi *et. al.* set up their own test environment and did not use DARPA/LL data set.

Multiple length N-grams

C. Marceau [20] reviewed the work of Forrest *et.al.* [18] and their N-gram characterization of sequences of system calls. She proposed an alternative characterization in the form of a Finite State Machine whose states represent predictive sequences of different lengths. An algorithm that constructs an FSM from training data is presented. In the presented approach the author first constructs a suffix tree for N-grams of the training data for some value of N that is large enough (small values of N lead to false negatives and large values of N increase training time). Then a suffix tree is converted to a DAG by merging equivalent subtrees. The result is a set of strings of varying length that is equivalent to the original set of N-grams. The contribution of the paper is that they show how to derive a FSM implementation of the sliding window algorithm presented in papers of Forrest *et.al.* [18] and reduce the size of data sets (177 strings and 1062 symbols from Forrest database are equivalent to 131 strings and 318 symbols from Marceau database).

DATA SETS

The author used `lpr` training data and exploits from UNM data set, `inetd` training data and exploits from UNM data set and data from the PersonalTracker applications collected by the CORBA Immune System. The CORBA Immune System uses the N-gram method to catch malicious programs that masquerade as legitimate clients.

Finite Automaton construction

Kosoresow [24] analyzed the structure of system call traces for normal and anomalous behavior. The audit trail used consists of traces of system calls executed by privileged processes. The parameters for each system call are ignored and the name of the call is replaced with a unique number. From that sequence they build a database of normal behaviors for each process. He counted the mismatches within small, fixed-length sections of the trace - *locality frames.* The frames were about 20 system calls long. In each position of the frame he counted the number of system calls that generated the mismatch. He observed that mismatches occur close together, which lead him to the conclusion that only short sequences of system calls are needed for detection. The other observation was that long sequences of system calls were consistently repeated, which lead him to use a Deterministic Finite Automaton (DFA) and its corresponding regular language. He constructed a DFA using manually selected macros (variable-length patterns of system calls). He divided the trace into the prefix, main and suffix portion and found the longest common prefixes and suffixes for each of the categories and looked for frequently occurring strings. He then substituted the strings for the macros and built a DFA from these letters. After applying this algorithm, out of 147 `sendmail` processes only 26 distinct processes were left.

The disadvantages of his approach were that he created DFA using scripts and by hand, which is not efficient and does not scale well to real systems. That leads to the conclusion that an exact DFA representation is not possible. Also, calculating macros based on the minimal description of the normal trace is potentially NP-hard. The observation that mismatches occur in occasional bursts does not hold for stealthy attacks.

DATA SET

Kosoresow used UNM data set.

Fast-automation based method

R. Sekar *et.al.* introduced a fast-automation based method for detecting anomalous program behaviors [25]. They criticize the approach of Forrest *et.al.* [17] because small values of N lead to the possibility of not capturing all behaviors and missing attacks. They take the Finite State Automaton (FSA) approach to the problem since an FSA can capture an infinite number of sequences of arbitrary length using finite storage. The authors claim that their FSA representation leads to performance better than the one in Forrest *et.al.* [17]. Their approach has the property that it learns faster, has better detection, false positive rate drops, presents large sets of execution traces in a compact way and has faster detection. Their learning algorithm is based on tracing the system calls made by a process under normal execution. In addition to system call names, they obtain the value of program counter (PC) as well since each value of the PC corresponds to a different state of the FSA. For each system call they obtain the corresponding location from where the call was made, check if there exists a transition from the current state to the new state that is labeled with the system call name that was intercepted (if not, there is an anomaly) and finally update the state of the automaton to correspond to the new state. The authors show that their algorithm converges quickly, false positive rate of the FSA algorithm is low and space and runtime overhead of FSA-learning is minimal.

DATA SET

Sekar *et.al.* [25] used `ftpd, httpd, nsfd` and `telnetd` programs and compared their approach to the N-gram approach on that data set. They compared the algorithms with respect to convergence of learning, false positive rate, runtime and space overhead and attack detection efficiency. They also compared the results using a live Web server. They argue that their system is able to detect almost all buffer overflow attacks, Trojan Horse and other code changes, maliciously crafted input, dictionary or password guessing attacks and DoS attacks. The FSA-based approach cannot detect attacks that involve system call argument values, race conditions (and other attacks that do not change behavior of attacked program) and certain classes of attacks launched with knowledge of the intrusion detection techniques being used.

Program Behavior Profiles

The work of Ghosh *et. al.* [26] is based on the work of the UNM group. The goal of their approach was to employ machine learning techniques that can generalize from past behavior so that they are able to detect new attacks and reduce the false positive rate. The first approach they used was equality matching. The database used was the normal behavior of programs. Instead of using `strace`, they use BSM data to collect the system call information and collect them into fixed size windows. They used the fact that the attacks cause anomalous behavior in clusters, like Kosoresow in [24]. Their decision choice looks for local clustering of the mismatches. The second model was the feed-forward topology with backpropagation learning rules. Database was not on per-user basis like in [14], but they built profiles of software behavior and malicious software behavior. During training many networks were trained for each program and the network that performed the best was selected. The training process consisted of exposing networks to four weeks of labeled data and performing backpropagation algorithm to adjust weights. They trained Artificial Neural Networks (ANN) to recognize whether small, fixed sequences of events are characteristic of the programs in which they occur. For each sequence, the ANN produces an output value that represents how anomalous the sequence is. The leaky bucket algorithm was used to classify the program behavior and using leaky bucket they made use of the rule that two close anomalies have higher influence than when they are apart. The final approach was Elman network, developed by Jeffrey Elman, since they wanted to add information about the prior sequences (DFA approach lacked flexibility and ANNs have the ability to learn and generalize). In order to generalize they employ a recurrent ANN topology of an Elman network. They train the network to predict the next sequence that will occur at any point in time. The Elman network approach had the best results, with 77.3% detection and no false positives and 100% detection and fewer false positives than in two other cases.

They also tried two approaches in which they used FSMs for their representation. Audit data was condensed into a stream of discrete events, where events are system calls recorded in data. Separate automata are constructed for different programs whose audit data are available for training. The training algorithm is presented with a series of $n-$grams taken from non-intrusive BSM data for a given program. The audit data is split into sub-sequences of size $n + l$ (n elements define a state and l elements are used to label a transition coming out of that state). The second FA-like approach, called a string transducer makes an attempt to detect subtler statistical deviations from normal behavior. It associates a sequence of input symbols with a series of output symbols. During training they estimate the probability distribution of the symbols at each state and during testing, deviations from this probability distribution indicate anomalous behavior. They use FA whose states correspond to $n-$grams in the BSM data. For each state they also record information about successor $l-$grams that are observed in the training data. During training their goal is to gather statistics about successor $l-$grams. They estimate the probability of each $l-$gram by counting.

DATA SETS

They used 1998 and 1999 DARPA/LL datasets. They used twelve weeks of training data from Lincoln Labs. They also used some additional data from Johns Hopkins University and they collected data for `eject, fdformat` and `xterm` on their own system. The data contained probes, DoS attacks, unauthorized accesses and unauthorized privilege elevations. The results show that under 94% of the attacks were detected. However, they did not include the attacks that did not misuse existing programs, meaning that the actual detection rate is much smaller.

2.1.4 Anomaly detection using data mining

The ADAM (Audit Data Analysis and Mining) system [30] is an anomaly detection system. It uses a module that classifies the suspicious events into false alarms or real attacks. It uses data mining to build a customizable profile of rules of normal behavior and then classifies attacks (by name) or declares false alarms. ADAM is a real-time system. To discover attacks in TCPdump audit trail, ADAM uses a combination of association rules, mining and classification. The system builds a repository of normal frequent itemsets that hold during attack-free periods. Then it runs a sliding window online algorithm that finds frequent item sets in the last D connections and compares them with those stored in the normal item set repository. With the rest, ADAM uses a classifier which has previously been trained to classify the suspicious connections as a known type of attack, unknown type or a false alarm. *Association rules* are used to gather necessary knowledge about the nature of the audit data. They derive a set of rules in form $X \to Y$, where X and Y are sets of attributes. There are two parameters associated with a rule: *support* s and *confidence* c. The definitions of s and c are as follows. The rule $x \to Y$ has support s in the transition set T if s% of transactions in T contain X or Y. The rule $x \to Y$ has confidence c if c% of transactions in T that contain X also contain Y. If the item set's support surpasses a threshold, that item set is reported as suspicious. The system annotates suspicious item sets with a vector of parameters. Since the system knows where the attacks are in the training set, the corresponding suspicious item set along with their feature vectors are used to train a classifier. The trained classifier will be able to, given a suspicious item set and a vector of features, classify it as a known attack (and label it with the name of attack), as an unknown attack or a false alarm.

DATA SET

ADAM was one out of 7 systems tested in the 1999 DARPA evaluation. In overall evaluation it performed about the same as EMERALD and better than all other systems. It was focused mainly on detecting DoS attacks and probes.

Tables 2.1 and 2.2 represent the main features of the presented techniques.

Authors	Approach	Database of normal	Profiles	Rules	Events	Signatures	Data set	Classific.
Lane, Brodley	Machine Learning	YES	YES	NO	NO	Actions	User generated	Normal-Abnormal
Teng, Chen, Lu	Time-based inductive learning	YES	NO	Sequential rules	YES	Patterns	User generated	Normal-Abnormal
Barbour	Program Modeling	YES	YES	NO	NO	NO	DARPA, UNM, own	Normal-Abnormal
Forrest, Warrender	Computer immunology	Privilelged processes	NO	NO	NO	YES	UNM	Normal-Abnormal
Wespi, Debar	Variable pattern length	YES	NO	NO	NO	YES	one privileged proc.	Normal-Abnormal

Table 2.1: Overview of Anomaly detection-based Intrusion Detection Approaches.

Authors	Approach	Database of normal	Profiles	Rules	Events	Signatures	Data set	Classific.
Marceau	Multiple length n-grams, FSM	YES	NO	NO	NO	YES	UNM, CORBA	Normal-Abnormal
Kosoresow	DFA manual	Privileged proc.	NO	NO	NO	System calls	macros	Normal-Abnormal
Sekar	Fast-automation based	YES	NO	NO	NO	System calls and PC	ftpd httpd nsfd telnetd Web server	Normal-Abnormal
ADAM	Association rules, classification	YES	YES	YES	NO	NO	DARPA	by name

Table 2.2: Overview of Anomaly detection-based Intrusion Detection Approaches.

2.2 Misuse Detection

2.2.1 Language-based Misuse Detection

Language-based Misuse Detection systems accept a description of the intrusions in a formal language and use this to monitor for the intrusions. Most languages for misuse systems, including the one used by NIDES, are low-level and have limited expressiveness.

ASAX

Habra *et. al.* define Advanced Security audit trail Analysis for universal audit trail analysis [32]. ASAX (Advanced Security audit trail Analysis on uniX) uses RUSSEL, a rule-based language, specifically appropriate for audit trail analysis. ASAX sequentially analyzes records using a collection of rules that are applied to each audit record. A subset of rules is active at any time. They define a normalized audit data format (NADF) as a canonical format of the Operating System's audit trail.

2.2.2 Expert Systems

Expert systems use lists of conditions that, if satisfied, indicate that an intrusion is taking place. The conditions are rules that are evaluated based on system or network events. These rules are specified by experts familiar with the intrusions, generally working closely with the developers of the system.

2.2.3 High-level state machines for misuse detection

The **State Transition Analysis Tool (STAT)**[33] describes computer penetrations as attack scenarios. It represents attacks as a sequence of actions that cause transitions that lead from a safe state to a compromised state. A state represents a snapshot of the system's security-relevant properties that are characterized by means of assertions, which are predicates on some aspects of the security state of the system. The initial state of a transition diagram does not have any assertions. Each state transition diagram starts with a signature action that triggers monitoring of the intrusion scenario. Transitions between states are labelled by the actions required to switch from one state to another. These actions do not necessarily correspond to audit records. The resulting state transition diagram forms the basis of a rule-based intrusion detection algorithm.
USTAT [34] is an implementation of the STAT tool developed for Solaris BSM. It reads specifications of the state transitions necessary to complete an intrusion and evaluates an audit trail. Two preconditions must be met to use USTAT: the intrusion must have a visible effect on the system state and the visible effect must be recognizable without knowing the attacker's true identity.

STATL [35] is a language that allows description of computer penetrations as sequences of actions that an attacker performs to compromise a computer system. The attack is modelled as a sequence of steps that bring a system from an initial safe state to a final compromised state. An attack is represented as a set of states and transitions, where each transition has an associated action. The notion of *timers* is used to express attacks in which some events must happen with an interval following some other event (set of events). Times are declared as variables using the built-in type `timer`. STAT can detect cooperative attacks and attacks that span multiple user sessions, can specify rules at higher level than audit records, is easier to create and maintain than other rule-based methods, can represent a partial ordering among actions, can represent longer scenarios than other rule-based systems. On the other hand, it has no general-purpose mechanism to prune partial matches of attacks other than assertion primitives built into the model.

DATA SETS

USTAT and NetSTAT were tested in the 1998 DARPA/LL off-line intrusion detection evaluation and the 1998 AFRL real time evaluation.

2.2.4 EMERALD

EMERALD (Event monitoring enabling responses to anomalous live disturbances) [31] employs both anomaly and misuse detection. It includes *service analysis* that covers the misuse of individual components and network services within a single domain, *domain-wide analysis* that covers misuse visible across multiple services and components and *enterprise-wide analysis* that covers coordinated misuse across multiple domains. They also introduce the notion of *service monitors* that provide localized analysis of infrastructure and services. EMERALD consists of three analysis units: profiler engines, signature engines and resolver. Profiler engine performs statistical profile-based anomaly detection given a generalized event stream of an analysis target. Signature engine requires minimal state management and employs a rule-coding scheme. The profiler and signature engines receive large volumes of event logs and produce a smaller volume of intrusion/suspicion reports and send that data to the resolver. The signature analysis subsystem allows administrators to instantiate a rule set customized to detect known problem activity occurring on the analysis target.

2.3 Specification-based approaches to Intrusion Detection

In specification-based approaches we don't need to specify signatures for known attacks, but we can instead specify known properties that should hold true in a valid execution. This approach is user insensitive, requiring manual specification of program behaviors.

2.3.1 Specification of legal activities

Ko *et.al.* [27] developed an approach using specifications of legal activities for security critical programs to verify program execution. They aim to detect exploitations of vulnerabilities in privileged programs by monitoring their execution using audit trails. The monitoring is with respect to specifications of the security-relevant behavior of the programs. Their approach involves the writing of specifications of legal activities for privileged programs and the use of these specifications to verify program execution. The specification language is based on *predicate logic* and *regular expressions*. The language alphabet consists of a set of operation predicate symbols OP, a set of attribute symbols A_t for each type $t \in T$ (T is the set of object types) and a set of state variable symbols S. The body of a program policy specification is a list of **rules** that characterize the set of parameter values allowed for each operation.

DATA SETS

Ko *et. al.* use a few attacks that exploit vulnerabilities in Unix privileged programs: `rdist, finger daemon` and `sendmail`.

2.3.2 Process Behavior Monitoring

R. Sekar [28] has presented an alternate approach to specification-based intrusion detection. This approach uses a domain-specific language called *behavioral monitoring specification language (BMSL)*. BMSL consists of rules in the form *pat* → *action*, where *pat* is a pattern on event sequences and *actions* specifies the responses when the observed history satisfies *pat*.

They developed generic specifications (grouped system calls of similar functionality), refined the generic specifications for `setuid` programs and `daemon` processes. Then they developed application-specific specifications for FTP and telnet servers and added site-specific security policies to these specifications.

DATA SET

1999 DARPA/AFRL online evaluation, 1999 DARPA/LL offline evaluation data and several locally developed experiments were used. They could detect 80% of the attacks using specifications that characterized legitimate program behaviors. They describe how their system was able to detect buffer overflows, ftp-write attack, warez attack, guess telnet, guest and HTTPtunnel attacks. They claim that no false alarms were reported by their method on the BSM data.

2.3.3 Process State Analysis

Nuansri *et. al.* [29] analyzed the pattern of system calls of a process, so that later the tracing technique could be used for an intrusion detection method. They used `ktrace()` system call to trace process activities. The tracing function reports all system calls used in a traced process as well as their arguments and return values, an error number and an error message (if they exist). They classified the user attributes into states, where state is described by a 4-tuple: real user ID, effective user ID, real group ID and effective group ID. States can be *normal, special privileged* and *superuser* states. The authors created a state transition diagram representing changes in process privilege where illegal transitions were used to create a set of rules (five rules) for use in intrusion detection.

This intrusion detection technique has a delay in detection because the `ktrace()` writes into a file which is then read by the intrusion detection system.

Table 2.3 represents the overview of presented Specification-based techniques. All classification methods are based on Normal/Abnormal classification.

2.4 Other approaches

2.4.1 Attack graph/tree approach

Automated generation and analysis

Sheyner *et. al.* [36] adopted the attack graph approach to attack detection. The whole process has three steps: modeling the network, producing the attack graph and analyzing the attack graph. They model the network as a FSM, where state transitions correspond to atomic attacks. They produce the attack graph using the model checker NuSMV [37]. Since the attack graph produced that way is a low level state transition diagram they parse the graph and reconstruct the original meanings of the state variables as they relate to the network intrusion. They model the network as a set of facts, each represented as a relational predicate. The state of the network specifies services, host vulnerabilities, connectivity be-

Authors	Approach	Database of normal	Profiles	Rules	Events	Signatures	Data set	Language
Ko, Fink, Levitt	Specification based	Privileged proc.	Specifications	NO	YES	Attributes	rdist, finger daemon, sendmail	YES
Sekar	Process beh. monitoring	YES	NO	YES	Actions	NO	DARPA and own	YES-BSML
Nuansri	State trans. diagram, process state analysis	YES	NO	YES - 5	NO	Patterns of sys calls	NO	NO
Sekar (DARPA evaluation)	Interception of sys calls, EFSA	YES	NO	YES	YES	Patterns	ftpd, telnetd, httpd	YES

Table 2.3: Overview of Specification-based Intrusion Detection Approaches.

tween hosts and a remote login trust relation. They suppose that the intruder has knowledge about the target network and its users, such as knowledge about host addresses, known vulnerabilities, information about running services, etc. Each node in the attack graph is labelled by an attack ID number (corresponds to the atomic attack to be attempted next), a flag that indicates whether the attack is stealthy or detectable and the numbers of the source and target hosts. As the authors claim, the bottleneck of their approach is the graph creation procedure since it took two hours to create a graph that had 3 hosts and eight attacks.

DATA SETS

They modeled sshd buffer overflow, ftp.rhosts, remote login and local buffer overflow.

Chapter 3

Attack representation and modelling

In this section we demonstrate that attacks have regular behavior that can be easily captured in four to five transitions. We present attack trees and their properties as means to model broad classes of network attacks. Comparing the transition diagrams of normal and malicious sequences we conclude that distinguishing among normal and malicious sequences is possible and can be done using as data sequences of system calls with information about payload size and shell execution and methods from detection theory.

This chapter presents also applications of attack tree models specifically to attacks exploiting buffer overflow attacks and of one Remote to Local attack. The model for `ftp-write` attack could not be applied for detection and classification due to lack of data. In this chapter we also present a description of race condition attacks. From that description we draw the conclusion that it is possible to detect and classify race condition attacks using our approach. The main goal of this thesis (as stated earlier) is to present efficient attack models and demonstrate their use for high performance detection and classification of attacks. Due to lack of attack data for other attacks we have shown results of application of our methods to attacks that exploit buffer overflows. To emphasize again, the main focus of this thesis is not modelling and detecting of buffer overflows, but modelling and detecting of broad classes of network attacks.

We are aware of the existence of tools for static code checking that are efficient in detecting buffer overflows and other attacks [43, 44, 42]. However, those tools have high false positive rates (up to 75%) and produce some false negatives (miss some attacks). Those tools are highly efficient if applied during the code development phase but it is not possible to apply them to already existing codes (or is very difficult to apply them). They also need implementation of certain rules and they demand a change in the code itself. Hence, it is possible to use them in the future and if used consistently they would significantly reduce the number of attacks that appear. The facts, however, show that only about 15% of code developers use programs that check code for possible errors like buffer overflows etc. and that proves that there is a need for dynamic (or on-line) Intrusion Detection systems now and in the future even for attacks that exploit buffer overflows. The need for IDS for attacks that exploit buffer overflows would be greatly reduced if all program developers used both static and dynamic code testing for buffer overflows.

In [42] the authors present a lightweight static analysis tool for static analysis. The tool is used for checking codes written in ANSI C. They show that the program generates both false positives and false negatives, meaning that it misses some of the attacks and also misclassifies normal behavior for anomalous. Their tool is based on reporting *all* suspicious commands like, for example, `gets` etc. Finally, the tool they introduced has extremely high false positive rate. Running the tool on wu-ftpd 2.5 produced 101 warnings. Out of those warnings, only 25 represented real buffer overflow attacks. Hence, the tool had FA rate around 75%. The authors claim that *"although static analysis is important approach to security it is not a panacea. It does not replace runtime access controls, systematic testing and careful security assessments"*.

Another paper that presents a technique for detecting potential buffer overflows by static analysis was written by David Wagner *et. al* [43]. Their approach involves synthesis of ideas from several fields, including program analysis, theory and systems security. They formulate the buffer overrun detection problem as an integer constraint problem and they use some simple graph-theoretic techniques to construct an efficient algorithm for solving the integer constraints. The tool the authors present produces both false positives and false negatives. The authors define the number of false negatives the tool produces as limitations of the tool and say that *"a human must still devote significant time to checking each potential buffer overrun"*. Hence, this version of the tool finds many of buffer overflows in tested programs but also misses some of them and there still exists a significant need for human presence.

David Wagner and Hao Chen present MOPS - MOdel Checking Programs for Security Properties in [44]. They identify rules of safe programming practice, encode them as safety properties and verify whether these properties are obeyed. The program to be modelled is represented as a pushdown automaton and the security property is represented as a Finite State Automaton. This tool also has high false alarm rate. There were no reports of False Negatives in this tool. The results are obtained on well known applications like wu-ftpd and sendmail.

In summary, if the static analysis tools are used for detection of buffer overflows they will detect the majority of attacks, but may also miss them. Every-thing is done under a high FA cost (around 75%). Those tools cannot detect other types of attacks successfully and there emerges a need for use of dynamic intrusion detection systems. In order to use static analysis tools efficiently all programmers have to use them when developing the code or re-apply the analysis tools to their already existing programs. That is infeasible and hence the need for dynamic intrusion detection systems still exists.

In this chapter we first give the definition of attack trees, then we describe each of the attack using the attack trees and finally we argue in support of the claim that attack trees are useful for understanding attacks but may not be appropriate as a main tool for attack detection. Then we present each of the attacks in the condensed form, containing only the states, transitions and parameters that characterize those attacks.

3.1 Attack representation

An attack tree [40, 41] is a Directed Acyclic Graph (DAG) with a set of nodes and associated sets of system calls for those nodes. Attack trees provide a formal, methodical way of describing the security of systems, based on various attacks. They give a description of possible sets of subactions that an attacker needs to fulfill in order to achieve a goal and bring the network to an unsafe state [41]. We represent attacks against the system in tree structure where the goal is to compromise the root node. Different ways of achieving the goals are represented in the form of leaf nodes. Each node/state has some associated memory and represents a goal towards achieving the final result. The values stored in the nodes should have the ability to update themselves with the new information about attacks. If we associate each edge in the tree with some probability/weight as *weight=f(configuration, attacker_profile)* then this information should be updated with new attacks. The individual goals can be achieved as sets of `OR` and `AND` sets of actions and subactions. Subactions are actions of the attacker that he needs to fulfill in order to achieve his goal (for example, in order to create a stealthy attack the attacker has to interleave some "normal" sequences of actions). The real challenge in constructing attack trees is that we must assume different levels of skills of the attackers, try to predict different paths of intrusions and evaluate costs of attacks. *States* are defined as actions performed by an attacker at a certain time. *Transitions* are caused by changes to a current state. We can further describe transitions by the property that transition from state n to state $n+1$ corresponds to an attack whose *preconditions* are satisfied in state n and whose *postconditions* hold in state $n+1$. Each transition involves the application of an operation that takes the stored value and input and generates output and a new value for the stored value. Using this notation we can describe an attack as a sequence of transitions that ends with the action of the attacker compromising the security of the system. Each node is a rule or a set of rules that gives us information about how the intruder can influence the network.

We have a malicious user who executes system calls in certain order to attempt to break in the computer system. The major problem is that in most cases of high-level attacks those sequences of system calls can be interleaved within normal system behavior in numerous ways. Our goal is to detect when a particular sequence of harmful instructions that could compromise security of our system has been performed. Though the distributed nature of these measurements is very important (as when an attack unfolds), we ignore it at present.

We need to represent a program as an attack tree, with goals and transition probabilities, as defined in [41] in the following way:

1. Create an attack tree representation where each node is a subgoal towards reaching the desired goal and each edge is labeled with the system calls made by the monitored program.
2. Estimate the attack tree in a probabilistic manner based on system calls made by the program.
3. Generate a database of actions from the given attack tree and convert those actions in a set of HMMs, where each HMM is trained to recognize the attack pattern from which it was generated.
4. Use the current state to rule out actions of the attacker with unmatched preconditions.
5. The IDS system performs pattern matching of HMMs and possible sequences of the attack and produces a score for each of the patterns.
6. Use the obtained scores to find the likelihoods of the attacker's actions that were generated.
7. Use the obtained data to determine if the program is being attacked.

The first step in analyzing the tree is to determine the set of states that are reachable from the initial state and find the unsafe states (paths that lead to compromising the state of the system). The first transition in the attack tree picks a subset of attacks that the intruder might use based on the set of preconditions. Each node in the level lower than the level of the root is a subgraph where another subset of attacks is used etc. This leads to the conclusion that in order to check every possible subset of attacks we need to run the check exponential number of times to the number of attacks.

3.2 Attack description and modelling

This section includes description of most commonly exploited categories of attacks and equivalent graphs that represent the flow of execution of those attacks. Finally, we define each attack with a sequence of 4-5 system calls and corresponding arguments. The studied attacks include four instances of User to Root attacks that exploit buffer overflows and two instances of Remote to Local attacks. The attacks studied are: `eject, ps, ffbconfig, fdformat` and `ftp-write`. The data used for attack detection was MIT Lincoln Labs 1998 data set for `eject, ffbconfig, fdformat` and `ftp-write`. Although the MIT Lincoln Labs attack database indicates that ps attack was detected both in 1998 and 1999, the table that contains attacks detected in 1998 does not contain it. Hence, the 1999 data was used to create a representation of that attack. The models for all other attacks were created using 1998 data and were also compared against 1999 data. The resulting models that were created from 1998 data could be used for attack detection from the 1999 data set as well and that verified the correctness of diagrams for 1998 data set. BSM data did not suffice for detecting `ftp-write` attack and for that reason both BSM and tcpdump data were used. BSM data were sufficient for capturing the behavioral pattern of exploited programs and tcpdump data were used to check whether the user initially logged in as root or obtained the root access after login. In the former case the privilege abuse was due to poor password or unchanged default root password and in the latter case it was due to the fact that the legal (non-root, in most cases anonymous) user used a security hole and obtained the root access after login process. Certain regularities were captured in behavior of exploited programs by comparing them against:

1. normal instances of those programs,
2. other instances of attacks detected at different periods of time and
3. by searching for certain events in the behavior of a program that were expected to happen by using the knowledge about the mechanism of those attacks and their goals.

All attacks except `ftp-write` had numerous instances that were performed throughout every week of data capturing which lead to the conclusion of existence of regularities in behavior. 13 instances of eject attack were captured, 3 instances of ps attack, 5 instances of ffbconfig attack, 5 instances of format attack and one instance of ftp-write attack (containing multiple exploits). All those behaviors completely matched the expected pattern learned by our theoretical approach.

The process of creating a diagram of attacks consisted of the following actions: the BSM data for specific days that contained the desired attacks were extracted in the first step. The second step consisted of extracting instances of those attacks using the program specially created for that purpose. Once the file with the attack was extracted, we looked at system calls along with their process numbers. As long as the critical program with the same process number was executed, we looked at the recorded system calls. When only system calls were not sufficient, we looked at the path of program execution (for example, in ftp-based attacks).

3.2.1 User to Root Attacks

User to Root exploits belong to the class of attacks where the attacker gains the access to a normal user account on the system and by exploiting some vulnerability obtains the root access. As mentioned before, the special class of those attacks are stealthy attacks that are more difficult to detect. According to MIT Lincoln Labs description of attacks, stealthy U2R attacks consist of six stages: encoding, transport, decoding, execution, actions and cleanup.

The most common User to Root attack is buffer overflow attack. Other, also commonly exploited attacks are loadmodule attack that exploits programs that make assumptions about the environment in which they are running, race condition attacks, etc. Some examples of those attacks are `eject, ffbconfig, fdformat, loadmodule, perl, ps, xterm` etc. and some of those attacks are described later in this chapter.

The following examples show that each attack is characterized with a very simple distinguishing sequence of system calls and accompanying parameters (like size, PID, path, etc.), which can be used for recognition and identification of different attacks.

Buffer overflow attacks

The most exploited type of User to Root attacks is the *buffer overflow attack*, which enables the attacker to run personal code on a compromised machine once the boundary of a buffer has been exceeded, giving him the privileges of the overflowed program (which in most cases is root). This type of attacks usually tries to execute a shell with the application's owner privileges.

A buffer overflow is the result of passing more data into a buffer than it can handle. As the stack contains not only the variables of vulnerable functions but also the return addresses for the points from which the functions were called, overwriting enough data potentially gives the exploit full control over the process. To achieve execution of exploit code, in most cases it is sufficient to write a shell code that spawns a shell. When we write or obtain the shellcode we know it must be part of the string which we are going to use to overflow the buffer. Also, the return address must point back into the buffer. After that the attacker is able to execute all other commands from the spawned shell.

Programs written in C are particularly vulnerable to buffer overflow attacks because C allows direct pointer manipulations without any bounds checking.

Two categories can be recognized: stack and heap based overflows. The simplest example of buffer overflow vulnerability is *stack smashing* ([38]), where the attacker overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address it jumps to the address that was placed on the stack by the attacker, making the application execute some arbitrary code. To provide the attacker with the root access, the application has to run under an ID that's different from the user's ID, like `suid` or `daemon`. To be able to execute a buffer overflow attack the attacker needs a shellcode. The most exploited function for calling a shell is `execve()`. If `execve()` succeeds, the calling program is replaced with the executable code of the new program and starts. Since the exploit is inserted in the attacked program we need to exit it as soon as the exploit code executes, which is done by using the `exit()` function.

The steps in executing a buffer overflow expoit can be summarized as follows:

- Fill the array `large_string[]` with the address of `buffer[]`, which is where the exploit code will be;

- Copy the shellcode into the beginning of the `large_string` string;

- Next, `strcpy()` copies `large_string` in the buffer without doing bounds checking. That action results in overflow of the return address, overwriting it with the address where our code is now located.

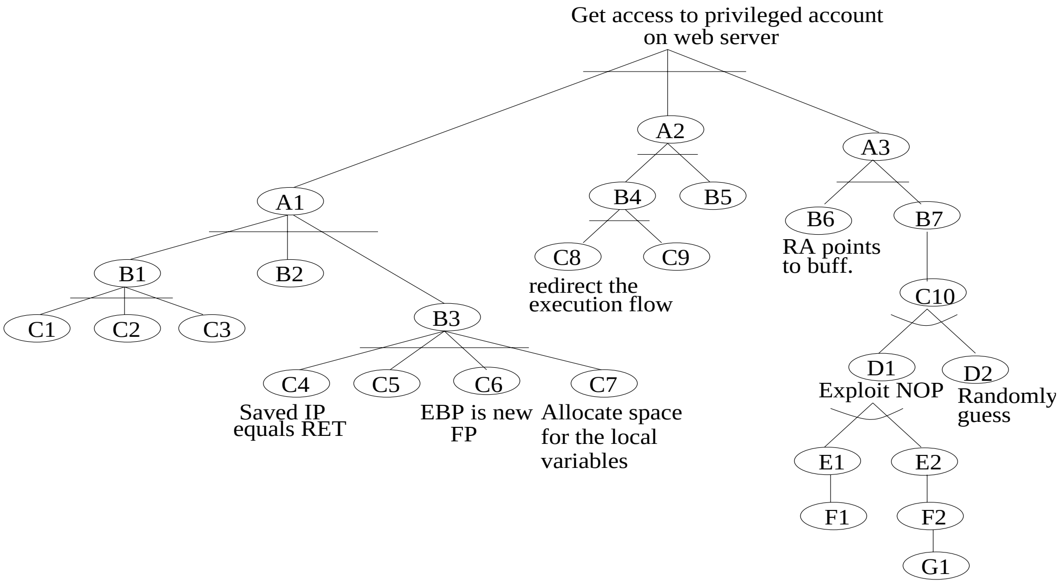
- Once we reach the end of main and it tries to return it jumps to the exploit code, and executes a shell.

The set of actions that lead to a buffer overflow attack can be graphically presented as shown on figure 3.1.

Eject attack - U2R

According to eject man pages, eject is used for those removable media devices that do not have a manual eject button, or for those that do, but are managed by Volume Management. The device may be specified by its name or by a nickname; if Volume Management is running and no device is specified, the default device is used.

The eject attack exploits a buffer overflow vulnerability in eject program. Due to insufficient bound checking on arguments in the volume management library, libvolmgt.so.1, it is possible to overwrite the internal stack space of the eject program. If exploited, this vulnerability can be used to gain root access on attacked systems.



A1. Identify the vulnerable program
A2. Write a shellcode − provide access to a privileged account
A3. Modify the flow of execution

B1. Call a procedure
B2. Push parameters on the stack
B3. Call function
B4. Make the shellcode part of the string we are going to use to overflow the buffer
B5. Copy large_string onto buffer
B6. Modify the return address
B7. Place the exploit code at the return address of a new program

C1. Save the previous FP so that it can be restored at the procedure exit
C2. Copy SP into FP
C3. Advance SP to reserve space for local variables
C4. Push the IP onto the stack
C5. Push the FP onto the stack
C6. Copy the current SP onto EPB
C7. Subtract size of local variables from SP
C8. Fill the array large_string[] with the address of buffer
C9. Copy the shellcode into the beginning of large_string[]
C10. Find out at which address our code will be

D1. Fill one half of the buffer with NOP instructions
D2. Use JMP and CALL instructions

E1. Return address points to the exploit code
E2. Return address points to the string of NOPs

F1. Execute the exploit code (1)
F2. Execute the NOPs

G1. Execute the exploit code

Figure 3.1: Buffer Overflow Diagram.

The MIT Lincoln Labs attack database gives some suggestions on how to detect that attack. Assuming that real attackers will not leave strings like “Jumping to address” or similar strings that would make detection trivial we need to extract certain patterns from traces of eject attack instances and observe all the regularities. We extracted numerous instances of both normal and anomalous sequences of eject program and noted regularities in program behavior. We also examined one instance of *stealthy* eject attack and noted the same regularities in program behavior as in clear instances of that attack. That can be explained with the fact that the stealthy part of the attack was performed in the login part and file transport to the exploited machine (i.e. encrypted file, scheduled exploit execution etc.) The attack traces consist of hundreds or thousands lines of system calls. However, there are only a couple of system call sequences that are sufficient to completely define the attack. The resulting state diagram has three states that uniquely define the attack. There are minor variations in the first stages of this attack and this resulted in one branch that was added to the diagram. One transition that enables detection of eject exploit is transition from `stat(2)` system call with path `/vol/dev/aliases` to `execve(2)`, whereas the normal sequence either does not have the same transition or has transition with *valid* path `/vol/dev/aliases/DeviceName` (for example `/vol/dev/aliases/floppy`) to `execve(2)`. The observed path `/vol/dev/aliases` with no device name after `aliases` indicates that the attacker is polling for existence of devices on the exploited network/machine and the string of `stat(2)` system calls preceding `execve(2)` system call defines the attack. No other attack has this identifying sequence. Another identifying string that characterizes the eject exploit is `usr/bin/eject` or existence of string `./ejectexploit` or `./eject`. Buffer overflow is detected by observing large value of header and a sequence of symbols such as ! @ # $ % &̂ . This may not be the case at all times because a skilled attacker may use a combination of letters and non-repeating patterns instead of repeating a short sequence of patterns.

The diagram of `eject` attack is presented on figure 3.2.

Another serious problem in attack detection is *globbing* which is used to avoid

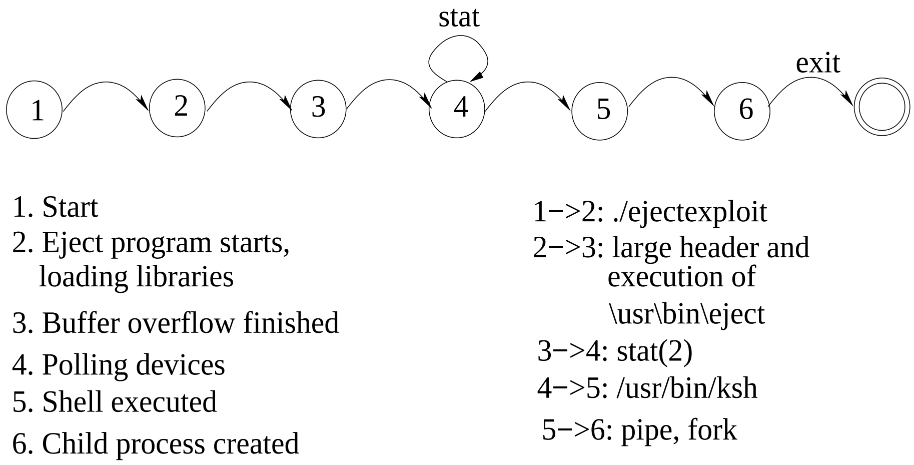


Figure 3.2: Eject attack.

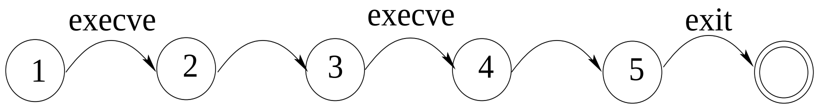
detection of suspicious commands. For example, instead of typing command `/bin/cat/etc/passwd` the attacker will use `/[r,s,t,b]?[l,w,n,m]/[c,d]?t /?t[c,d,e]/*a?s*`. The shell will replace the glob characters and will find that the only valid match for this string is `/bin/cat/etc/passwd`. None of the attacks from MIT dataset had examples of globbing.

Ffbconfig - U2R

The `ffbconfig` program configures the Creator Fast Frame Buffer (FFB)Graphics Accelerator, which is part of the FFB Configuration Software Package, SUNWffbcf. This software is used when the FFB Graphics accelerator card is installed. Due to insufficient bounds checking on arguments, it is possible to overwrite the internal stack space of the ffbconfig program.

This attack is another example of a buffer overflow attack that has signature very similar to the signature of `eject` attack. There is one state less in the equivalent representation than in the state diagram constructed for eject attack. The state that does not characterize `ffbconfig` exploit is `stat(2)` with undefined parameters. The sequence that defines this exploit is `/usr/sbin/ffbconfig` with oversized `-dev` parameter and `./ffbconfig` or `./ffb` or some other sequence that executes the exploit.

The equivalent state diagram is represented on Figure 3.3.



1. Start
2. Ffbconfig program starts, loading libraries
3. Buffer overflow finished
4. Shell executed
5. Child process created

1−>2: ./ffbconfig
2−>3: large header and execution of /usr/bin/ffbconfig
3−>4: shell execution: /usr/bin/ks
4−>5: pipe, fork

Figure 3.3: Ffbconfig attack.

Fdformat - U2R

`fdformat` attack is another example of buffer overflow attack. The Fdformat attack exploits a buffer overflow in the 'fdformat' program distributed with Solaris 2.5. The fdformat program formats diskettes and PCMCIA memory cards. The program also uses the same volume management library, libvolmgt.so.1, and is exposed to the same vulnerability as the eject program.

This exploit is almost identical to `ffbconfig` attack. The only differences are in the path of the attack which is in this case `/usr/bin/fdformat` and in the file that is executed (if it is named by the attack) `./formatexploit` or `./format`.

The equivalent state diagram is represented on Figure 3.4

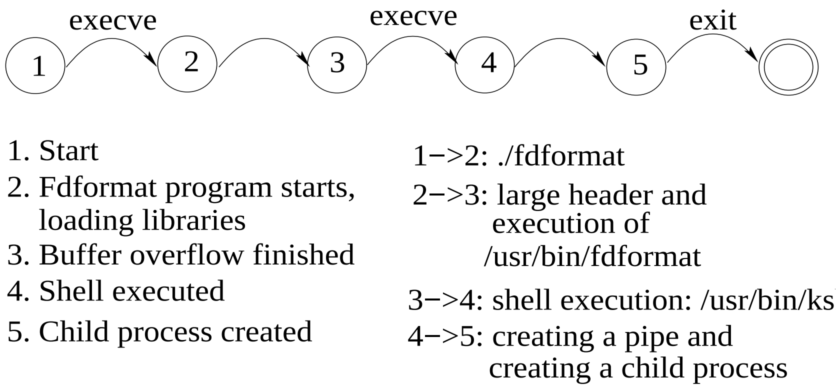

execve
execve
exit
1
2
3
4
5
1. Start
2. Fdformat program starts, loading libraries
3. Buffer overflow finished
4. Shell executed
5. Child process created
1−>2: ./fdformat
2−>3: large header and execution of /usr/bin/fdformat
3−>4: shell execution: /usr/bin/ks
4−>5: creating a pipe and creating a child process

Figure 3.4: Fdformat attack.

Ps attack - U2R

The typical form of ps attack involves both buffer overflow and race condition. The instances of ps attacks given in the MIT Lincoln Labs data set contain ps attacks that contain buffer overflow without race condition. This section contains the description of the ps attack that contains both buffer overflow and race condition. Finally the ps attack that is executed using only buffer overflow is described and represented with an equivalent state diagram.

The `ps` attack takes advantage of a race condition in the version of 'ps' distributed with Solaris 2.5 and allows an attacker to execute arbitrary code with root privilege. This race condition can only be exploited in order to gain root access if the user has access to the temporary files. Access to temporary files may be obtained if the permissions on the /tmp and /var/tmp directories are set incorrectly. Any users logged into the system can gain unauthorized root privileges by exploiting this race condition.

The ps program has to see the information about all processes currently running on the machine, so the program has to run as root. The first time ps is run on Solaris, it looks up the location of the process table inside the kernel and other details it needs. Searching for this information is time-consuming, it stores it in a file called `/tmp/ps/data`, where future invocations of ps can find it. The assumption was that the user could delete only files that belong to him in `/tmp` directory, but that was not the case. Any user was able to delete other user's files. The actions `ps` performs are:

- first put the information in a file called `/tmp/ps.XXX` (XXX is the process ID of ps);
- change the owner of the file to root;
- rename the file to `/tmp/ps/data`.

The attacker needs to write an exploit program that deletes the `/tmp/ps/data` file. That action forces the ps program to create a new file and look in the `/tmp` directory for a file starting with ps. When it finds the file, it deletes and replaces it with a symbolic link to another file. The attacker will probably be forced to run the exploit many times before a success occurs. When the exploit is successful the ps will perform `chown` command on the symbolic link. The result is that the file the link points to is owned by root.

The `ps` attack described above is ps *race condition* attack that does not include buffer overflow. `Ps` attack included in the MIT Lincoln Labs data set is based on buffer overflow and has signature almost identical to previously described attacks. It can also be detected by looking at large values of header and path of execution which is `/usr/bin/ps` in this case.

The resulting diagram is represented on Figure 3.5.

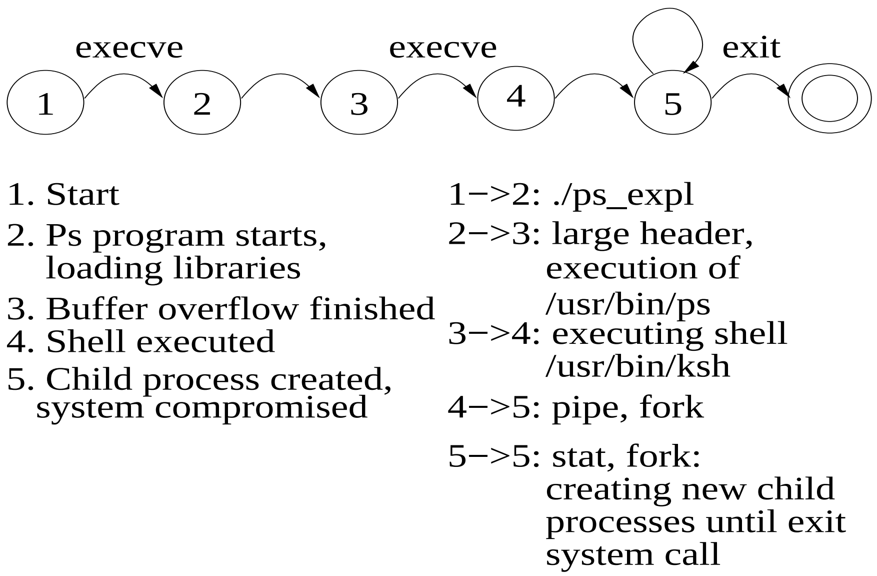

execve
execve
exit
1
2
3
4
5
1. Start
2. Ps program starts, loading libraries
3. Buffer overflow finished
4. Shell executed
5. Child process created, system compromised
1−>2: ./ps_expl
2−>3: large header, execution of /usr/bin/ps
3−>4: executing shell /usr/bin/ksh
4−>5: pipe, fork
5−>5: stat, fork: creating new child processes until exit system call

Figure 3.5: Ps attack.

Although not used in the experiments for attack detection, we present the description of format string attacks. According to many sources, format string attacks are much easier to detect than buffer overflows. We also describe race condition and stealthy attacks.

3.2.2 Format String Attacks

Just like buffer overflows, Format String Attacks are the result of bad programming practices and are widely exploited. They are not as difficult to detect as buffer overflows, but are dangerous if not detected on time. A typical example of possible format string attack is when a programmer uses: `printf(str);` instead of `printf("%s", str);`. Among many possible format commands the most dangerous from the point of view of format string exploits is $\%n$. The definition of this command from the printf() man page is: *The number of char-*

acters written so far is stored into the integer indicated by the int (or variant) pointer argument. %n assumes that the corresponding argument to `printf` is of type `‘‘int *’’` and writes back the number of bytes formatted so far. If the attacker carefully chooses the format string then he can use the %n directive to write an arbitrary value to an arbitrary word in the program's memory. The attacker can send a single packet of data to a vulnerable program and obtain a remote (possibly root) shell. Because %n treats the corresponding argument as an `int *` an effective format bug attack must go towards the top of the stack (by inserting some number of %d directives) until it reaches a suitable word on the stack. That word is treated as an `int *` and the attacker uses a %n to overwrite a word nearly anywhere in the victim program's address space. The attacker always provides a format string that does not match the actual number of arguments presented to `printf`.

3.2.3 Race condition attacks

By far the most dangerous User to Root Attack is the exploit of race conditions. Most of race conditions are exploited in cases when `setuid` to root program saves data in a file owned by the user executing the program. Three race condition exploits are described in this section: race condition in managing `tmp` files, `binmail` race condition and stealthy ps attack that includes buffer overflow and race condition attack. The DARPA data set did not have any examples of ps attack with race condition. The ps attacks presented there had only buffer overflows that were used to gain root access.

Race condition in managing tmp files

This is a simple example of race condition in Unix environment. As a result of this exploit, the attacker obtains access to the `passwd` file. It exploits the weakness in some versions of Solaris workstations in which any user can write and delete files in tmp directory. The same fact is later exploited in a more developed version of this attack that exploits race condition and buffer overflow. This simple race condition can be presented in the following steps:

- `/tmp/file` and `/etc/passwd` refer to different objects, where `file` is the file in `/tmp` directory;
- The attacker deletes the file from `tmp` directory before the process makes the system call which will open the file;
- Hard link for `/etc/passwd` is created (`/tmp/file`);
- When `open` system call wants to open the `/tmp/file` file it actually follows the newly created hard link and opens the `/etc/passwd` file;
- As a result of this exploit, the attacker can alter the password file and gain access to the system.

The following figure represents the flow graph of race condition attack:

Binmail Race condition

The `binmail` program delivers mail by writing it into the recipient's mailbox. This race condition is exploited in the following way:

- `lstat(2)` system call is used;

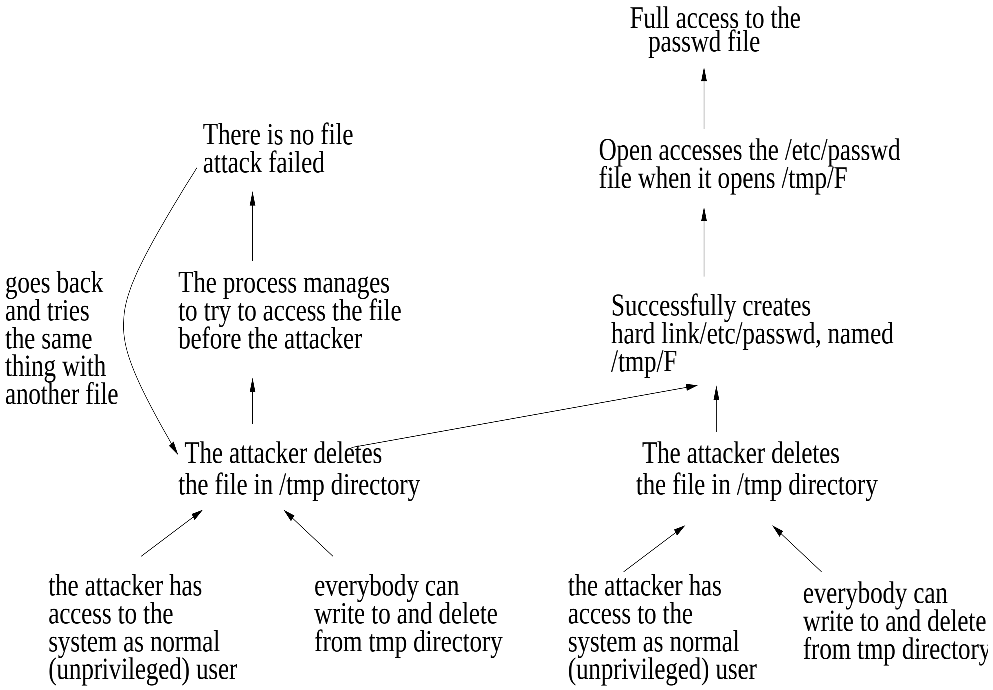

Full access to the passwd file
There is no file attack failed
Open accesses the /etc/passwd file when it opens /tmp/F
goes back and tries the same thing with another file
The process manages to try to access the file before the attacker
Successfully creates hard link/etc/passwd, named /tmp/F
The attacker deletes the file in /tmp directory
The attacker deletes the file in /tmp directory
the attacker has access to the system as normal (unprivileged) user
everybody can write to and delete from tmp directory
the attacker has access to the system as normal (unprivileged) user
everybody can write to and delete from tmp directory

Figure 3.6: Diagram of race condition attack.

- If the mailbox is not a symbolic link append the letter to mailbox as root;
- Attacker deletes the mailbox file;
- The attacker creates a new file with the same name, which is a link to the system password file;
- The letter will be appended to the password file

The following figure represents the binmail race condition attack:

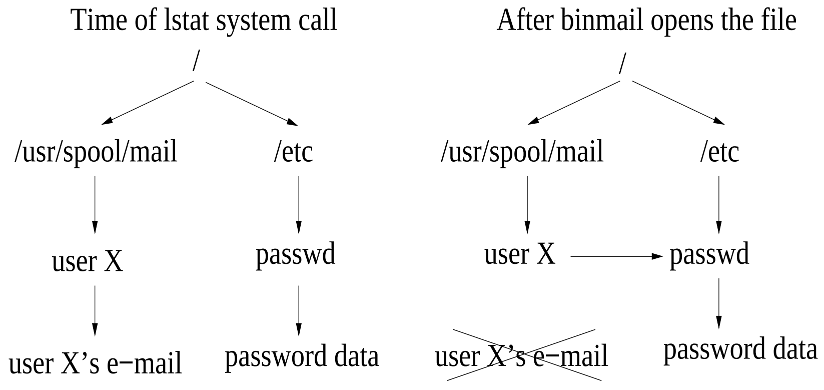


Figure 3.7: Diagram of binmail race condition attack.

3.2.4 Stealthy Ps Attack - Race condition and buffer overflow

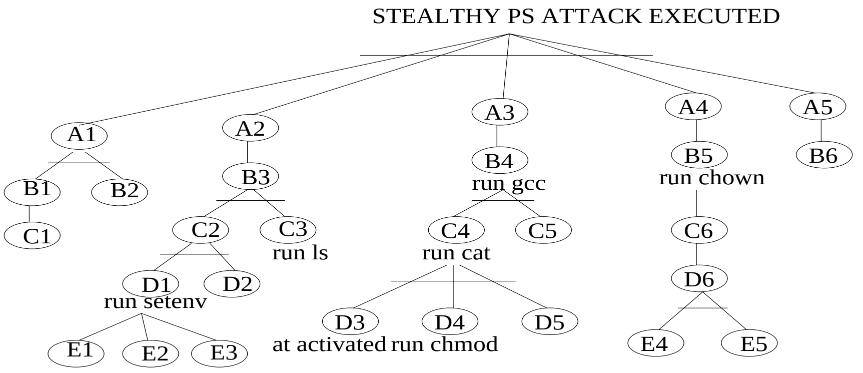
The `ps attack` takes advantage of a race condition in the version of *ps* distributed with Solaris 2.5 and allows an attacker to execute an arbitrary code with root privilege. This race condition can only be exploited to gain root access if the user has access to the temporary files. According to the CERT web page the ps program contains a vulnerability that does not sufficiently check the arguments passed to it, so it is possible to overwrite the internal data space of this program (the stack) while it is executing. To gain access to temporary files, permissions for the `/tmp` and `/var/tmp` directories have to be set incorrectly. When an attacker transfers his shell script to the attacked machine and runs it, a root shell will be spawned.

The ps program has to see the information about all processes currently running on the machine, so the program has to run as root. The first time ps is run on Solaris, it looks up the location of the process table inside the kernel and other details it needs. Since searching for this information is time-consuming, it is stored in `/tmp/` directory in file called `ps_data` (hence the full path is `/tmp/ps_data`), where future invocations of ps can find it. The assumption is that the user can delete only files that belong to him in `/tmp` directory, but that is not the case. Any user is able to delete other user's files. The actions ps performs can be described as follows:

- Put the information in a file called `/tmp/ps.XXX` (XXX is the process ID of ps);
- Change the owner of the file to root;
- Rename the file to `/tmp/ps_data`.

The attacker needs to write an exploit program that deletes the `/tmp/ps_data` file. That action forces the ps program to create a new file and look in the `/tmp` directory for a file starting with ps. When it finds the file, it deletes and replaces it with a symbolic link to another file. The attacker will probably be forced to run the exploit many times before the success. When the exploit is successful the ps will perform `chown` command on the symbolic link. The result is that the file the link points to is owned by the root.

The following diagram represents a **stealthy ps attack** that performs a number of additional actions to ensure the maximum secrecy and to decrease the probability of detection. Currently it is very difficult to detect stealthy attacks, especially race conditions because they happen during a long period of time and use masquerading, delayed execution of commands, encryption etc. The following figure represents the diagram of Ps attack:



A1. Setup phase finished
A2. Transport phase finished
A3. Execution phase finished
A4. Actions phase finished
A5. Cleanup phase finished

B1. Attacker performs normal commands
B2. Schedules future script execution using at command
B3. Extract the files
B4. The attack script compiles the exploit
B5. Restore the permissions of secret files
B6. Remove the files

C1. The attacker logs on
C2. Download the *tar file using the browser
C3. List the contents of the directory
C4. Create the machine code to overwrite the buffer
C5. Format the machine code into a message object
C6. Run other exploits

D1. Connections established
D2. Adjust display from victim to attacker
D3. Create sh and tcsh
D4. Make attack scripts executable
D5. Suppress output of the script execution
D6. Sleep

E1. Connect to the victim machine
E2. Establish X Windows connection back to the attacker
E3. Establish an HTTP connection
E4. cat and redirect output
E5. Move a secret file to another locatioi

Figure 3.8: Diagram of Ps attack.

3.2.5 Remote to User Attacks

It has previously been described what Remote to User attacks are and some typical examples were presented. Some examples of R2U attacks are the Dictionary, Ftp-Write, Guest, Imap, Named, Sendmail attacks etc.; they exploit abuse of feature, misconfiguration or bug to obtain access to the local host. Dictionary and Guest attacks attempt to break into the system either by guessing badly configured passwords or using the fact that on many systems there are at least a few guest accounts where the user can login without any password or a password like "guest", "anonymous" or no password at all. This type of attacks can be avoided by setting an appropriate threshold that will raise the alarm if there are unsuccessful attempts of logging into an account.

Ftp-write attack sequences were extracted from the MIT Lincoln Labs data set using the same procedure as described in the previous section. This is an attack where the anonymous ftp misconfiguration is abused. If the ftp server is misconfigured in the way that a remote user can add files, the attacker can add his own .rhosts file and log in again as user ftp. General sequence of executions in ftp-write attack is presented on the following diagram:

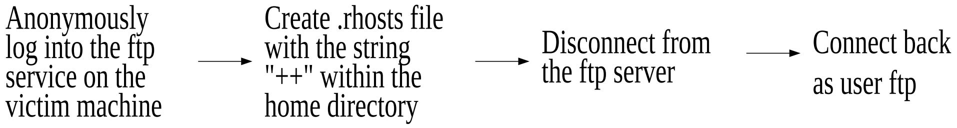

Anonymously log into the ftp service on the victim machine
Create .rhosts file with the string "++" within the home directory
Disconnect from the ftp server
Connect back as user ftp

Figure 3.9: Diagram of Ftp-write attack.

The mechanism of detecting Ftp-write attack is different from the previously described attacks. The ftp login session can last over a long period of time and that represents an obstacle in detecting an attack from memory point of view. However, traces of an ftp-write attack given in MIT Lincoln Labs data set indicate that the critical points that positively define the attack are observed in the initial parts of the trace. After the first execve system call the ./rhosts file is opened for reading, its status is checked, then its opened for writing, a permission to create a file is set and then a symbolic link is established. After that the attacker logs out and in the next login session it logs in as root. The following graph in Figure 3.10 represents general goals for this type of attack:

The login phase of the attack, where the attacker logs in to anonymous ftp

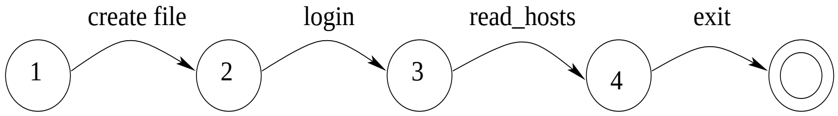
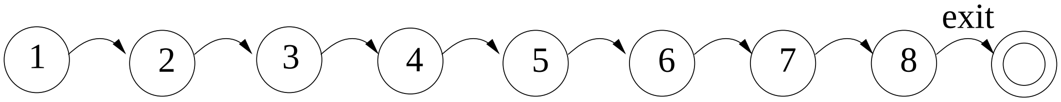


Figure 3.10: Ftp-write attack.

server and changes `.rhosts` file so that it can login as root next time is presented on Figure 3.11.



1. Start
2. Anonymous login succeeded
3. Status of .rhosts file obtained
5. File opened for write, create and truncate
6. Pipe created
7. Ownership of the file changed
8. File created

1−>2: login
2−>3: stat(2) on ftp/.rhosts
3−>4: stat(2) on /etc/shadow
4−>5: open(2) ftp/.rhosts
5−>6: open for read and write /ftp/dev/tcp
6−>7: chown(2)
7−>8: close(2)

Figure 3.11: Ftp-write attack - create file phase.

Chapter 4

Detection and classification algorithms for Hidden Markov Models

The main contribution of this thesis is an attempt to detect and classify different types of network intrusions (attacks). This chapter presents a short overview of techniques and general models used for HMMs and then it presents one detection algorithm and several classification algorithms (we emphasized on of these).

4.1 Introduction

We need to recognize a malicious user who executes system calls in certain order in attempt to break into the computer system. In most cases of high-level attacks, sequences of system calls can be interleaved within normal system behavior in numerous ways.

Our goal is to detect when a particular sequence of harmful instructions that could compromise security of our system has been performed.

The following plan for the recognition of malicious activity is proposed:

1. Generate a database of malicious actions and convert those actions in a

set of HMMs, where each HMM is trained to recognize the attack pattern from which it was generated.

2. Use the current state to rule out possible actions of the attacker due to unmatched preconditions ("and" sets of subconditions are not yet fulfilled);

3. The IDS performs matching of HMMs against possible attack sequences and produces a score for each of the patterns;

4. Use obtained scores to find the likelihoods of the attacker's actions that were generated and construct a vector that consists of probabilities that the user is performing each of recorded actions, i.e. that the user is actually the attacker.

There are several approaches that can be applied to the problem and each of them brings its own difficulties. The specific steps described above may differ slightly among different approaches. The notation that we are going to use is as follows. We denote our set of attack models as $\Lambda = \lambda_1, \ldots, \lambda_m$. To recognize an attack we need to compute the probability of seeing the observation sequence $O = o_1, \ldots, o_n$ given each attack model λ_i. We consider $O = (o_1, \ldots, o_T)$ to be the observed data and the underlying state sequence $q = (q_1, \ldots, q_T)$ to be hidden or unobserved.

4.2 Hidden Markov Models

Before proceeding to our problem formulation and proposed solutions we present the overview of notation used for HMMs. The notation in this thesis corresponds to the notation used in [39].

A discrete HMM is specified by the following triple

$$\lambda = (A, B, \pi) \tag{4.1}$$

A represents the state transition matrix of the underlying Markov chain and is defined as (where s_t denotes the state of the Markov chain at time t):

$$A = [a_{ij}] = [p(s_{t+1} = j \mid s_t = i)], i, j = 1, \ldots, N.$$

B is the observation matrix and is defined as (where x_t is the output (observation) at time t):

$$B = [b_{ij}] = [p(x_t = j \mid s_t = i)], i = 1, \ldots, N; j = 1, \ldots, K.$$

π is the initial state probability distribution of the underlying Markov states and is defined as

$$\pi = [\pi_i = p(s_1 = i)], i = 1, \ldots, N.$$

Given appropriate values of N, K, A, B and π the HMM can be used as a generator for an observation sequence. In our case we suppose that the number of states N and alphabet of observations K are finite. The joint probability of an HMM sequence of length n is

$$p(s_1, \ldots, s_n, x_1, \ldots, x_n) = \pi_{s_1} \left[\prod_{t=1}^{n-1} a_{s_t s_{t+1}} \right] \left[\prod_{i=1}^{n} b_{s_t x_t} \right].$$

To be able to use the algorithm that is presented next we need to compute the likelihood of a sequence of HMM observations given λ, $P(X \mid \lambda)$, which is done by using the "forward variable". According to definitions in [39] the forward variable of an HMM is defined as

$$\alpha_t(i) = p(x_1, x_2, \ldots, x_t, s_t = i \mid \lambda) \tag{4.2}$$

Hence, $P(X \mid \lambda)$ is the sum of $\alpha_t(i)$'s. Recalling the solution of the forward part of the forward-backward procedure [39], $\alpha_t(i)$ can be computed recursively. The initial condition in the forward procedure is

$$\alpha_1(j) = \pi(j) b_j(x_1); 1 \leq i \leq N \tag{4.3}$$

and accordingly the recursive step [39] is

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(x_{t+1}); 1 \leq t \leq T-1; 1 \leq j \leq N \tag{4.4}$$

where $b_j(x_{t+1})$ is the conditional density function given the underlying Markov state at time t is j.

We suppose that we have M HMMs as possible models for the observed data and we need to test which of the M HMMs matches the incoming sequence. In the framework of intrusion detection the problem can be formulated as follows: *given M attack models in the form of Hidden Markov Models with known parameters, detect the one that matches the incoming sequence with the highest probability.* However, in the case of detecting an attack the incoming sequence may or may not match one of the HMM models of attack. In case it does not match one of the attack models we need to consider two cases: either the incoming sequence is not an attack or it is an unknown attack that we don't have in our database. In this report we assume that the incoming HMM matches one of M HMMs in the system.

The problem of M-ary detection is solved with calculating log-likelihoods for each of the possible models given the observed sequence and finding the maximum. The model with the maximal log likelihood (closest to zero) wins and the attack is classified accordingly. In this thesis M=5. The likelihood function is calculated as follows:

$$f(x_1, x_2, \ldots, x_t \mid \lambda_l) = \sum_{i=1}^{N} \alpha_t(i) \tag{4.5}$$

for $l = 1, ..., M$. Here N represents the total number of states and the α_ts are the forward variables.

4.3 Algorithm

4.3.1 Data set

Experiments are performed using the 1998 and 1999 DARPA/LL offline evaluation data. Both data sets had to be used due to the fact that there are no examples of `ps` attack in the 1998 data set. Lincoln Labs recorded program behavior data using the Basic Security Module (BSM) of Solaris. We used BSM audit records that correspond to system calls. Each record provides information like name of the system call, a subset of arguments, return value, process ID, user ID etc. We used the system call information and information about the payload size associated with each system call.

In order to use the BSM data we had to develop an environment that would parse the BSM audit logs into a form that can later be used for detection and classification of attacks. The information about payload size and execution of an unexpected program was used for attack detection. Hypotheses testing was used for attack classification.

4.3.2 Data processing phase

Step one

The initial step deals with the whole BSM file that contains the daily activity performed on machine `pascal` and contains both normal and abnormal activity. Since many of the processes are interleaved, the original BSM file most probably has attacks interleaved within normal processes and which can be distinguished only by looking at the process numbers. In the initial step we divide the whole BSM file into chunks of length of 100 system calls. Each system call is assigned a number according to the order of appearance in the BSM sequence. The first couple of lines in the BSM file are system-related and we don't include them in any of the chunks.

As the result of phase one we have (total number of system calls in the BSM sequence)/100 files named WeeknumberDay.bsm.txt.split.i, each of which contains 100 enumerated system calls. In order to differentiate among abnormal and normal behavior we ran another algorithm. Denote one randomly chosen system call number as k. The algorithm observes the payload size associated with each system call. If the system call has payload greater than 300, the program outputs $2 * k$. Otherwise, it outputs $2 * k - 1$. The first step of the algorithm outputs another file that contains the total number of system calls, their names and associated numbers. It lists the even numbers that appeared in the BSM file (even numbers=system calls with high payload). In addition to observing the oversized arguments, the algorithm also monitors the traces in BSM files for violations in the form of unexpected program execution (shell). If both conditions are fulfilled (oversized argument and execution of unexpected program) there is an ongoing attack. The testing phase detects the type of the attack and classifies it.

The outputs of the first run are:

1. Total number of system calls;
2. Total number of different system calls;
3. List of enumerated system calls with shell execution in order of appearance in the BSM file;
4. List of even numbers in the files (system calls that have payloads greater than a threshold, in our case 300). We use this information in creating the B matrix for HMM training in later stages of the algorithm.

Step 2

This step takes the original BSM file as an input and looks for instances of each attack (if we want to have instances of 4 attacks, we have to run the program four times to get the result). This step is not computationally intensive as the previous one. Step one takes from 45min to more than 2h, depending on the length of BSM file and step 2 takes a couple of minutes to run. It produces meta-files WeeknumberDay.AttackName.txt containing line numbers of system calls issued by observed programs and the actual system calls with all parameters from the original BSM file.

The MIT data provides the information regarding the attack distribution throughout 5 weeks of training and 2 weeks of testing. The second week of training was not usable by the detection procedure since it contains only `tcp` data. Week 6 contained only `ffbconfig` and `eject` attacks. Week 7 could not be used since it contained only `tcp` data. Hence, the only attacks where multiple hypothesis testing approach was possible were `ffbconfig` and `eject`.

Step 3

The last step extracts sequences of system calls of length 100 and labels files that contain bad sequences (for example, ffb-bad.txt or format-bad.txt) using the files created in the previous step and the original BSM file. The resulting file contains system calls issued by the attacked program and some other system calls from other programs due to the fact that the divided files may (and probably will) catch parts of other programs.

As a result we have the original BSM file chunked into sequences of length 100 system calls, one file (meta file) for each attack that contains the lines from BSM file that contain execution of programs of interest (ffbconfig, format, eject or ps) and meta-files chunked into sequences of 100 system calls. Not all of those meta-files are bad. To decide which files are going to be used for training we use information provided by MIT Lincoln Labs that precisely states at what time the attacks are executed. One potential problem can appear in this approach and is the only cause of misclassification among different types of attacks. Training is performed on chunks that start exactly when the attack starts and testing is performed on sequences obtained by chunking the original BSM file. It may happen that the testing files (first and last) contain a large portion of normal sequences, which causes that the structure of training and testing file may significantly differ. That problem may be solved by sliding window approach on training sequences, but on the other hand, this approach increases the number of staring models and increases the complexity of the model. Two possible solutions of misclassification among different types of attacks are described in more detail in the following section.

4.3.3 Training

In this thesis we assume that the attack models are already known. The goal is to detect and classify the incoming sequence using the models generated on already known attacks. We trained the HMMs on attack sequences generated in the third step of the algorithm presented in previous sections. Each of the sequences is sequentially loaded and parameters are generated using the HMM toolbox in Matlab. We initially chose to have 5 observations and 10 iterations in training. B matrix size is 5x2*k and A matrix is 5x5. We initially chose all even columns of the B matrix to be equal to zero, except the ones that correspond to the system calls with high payload and all odd columns are greater than zero and generated by the Matlab funcion mk_stochastic (for example if even system calls that appear in the sequence are execve (92), open (24) and ioctl (88), all even columns in B matrix except 24, 88 and 92 will be equal to zero). Matrix A is chosen to be diagonal and is generated by using the *mk_stochastic* function from the toolbox.

Each attack is represented with 4-6 sequences of length 100 but only one or two of them contain the actual attack (depends on whether the same sequence contains oversized argument and shell executions or not).

After sequentially loading the attack sequences and training parameters on them in 10 iterations we will get 10 different log likelihoods for each model. Since the initial parameters for A and π matrices are created using Matlab function mk_stochastic and parameters of B matrix are adjusted to the model up to a certain point the initial estimation of log likelihood is low (first iteration). After several iterations the parameters of trained HMM are found and A, B and π matrices don't change any more. An example of log-likelihood estimation from training on `eject` attack from Week 5 Tuesday is presented in figure 4.1:

The training procedure consists of training each of malicious sequences on the same input parameters A_{in}, B_{in} and π_{in}, producing an output HMM with parameters π_i, A_i and B_i that characterize that specific malicious sequence (i denotes one of possible HMM models and varies in different training sets, but is usually less than 10). This property of HMMs that each set of parameters π_i, A_i and B_i fits a specific malicious sequence is used in the testing phase for classification of attacks in two levels: normal/abnormal and attack1/attack2. Training was performed on MIT Lincoln Labs data from weeks 1 to 5. The training data used in this thesis is presented in table 4.1.

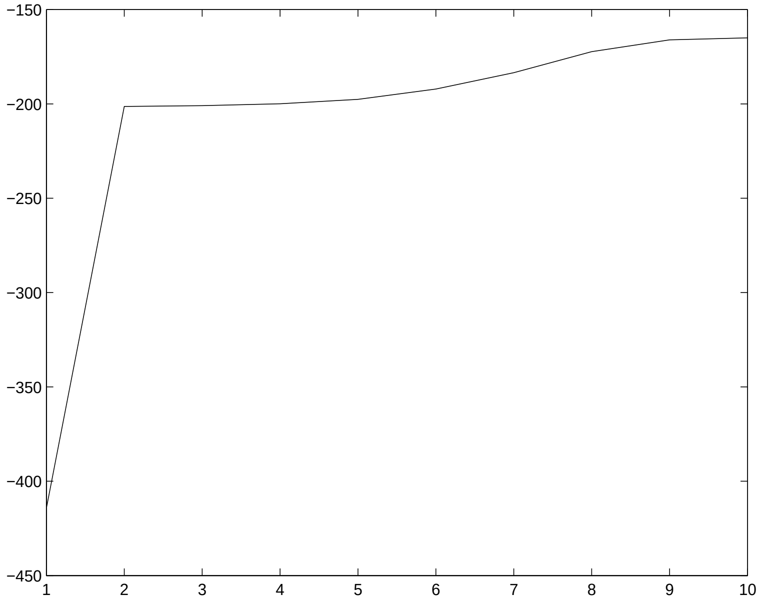


Figure 4.1: Procedure of parameter adjustment during HMM training.

Week	Day	Attack name	Variant
1	Monday	format	clear
1	Monday	ffb	clear
3	Monday	ffb	clear
4	Friday	ffb	add .rhosts, stealthy
4	Friday	format	stealthy
5	Monday	ffb	ftp's over exploit files
5	Monday	ffb	chmod files
5	Monday	ffb	executes attack
5	Monday	format	clear
5	Tuesday	eject	clear
5	Tuesday	eject	clear
5	Wednesday	eject	stealthy
5	Friday	eject	run self encoded exploit

Table 4.1: MIT Lincoln Labs training data sets

4.3.4 Detection

We concentrated on four programs: `ffbconfig, format, eject` and `ps`. The attacks on `eject, fdformat` and `ffbconfig` exploited a buffer overflow condition to execute a shell with root privileges. The goal was to create hypotheses for each of the attacks and classify the attacks in the appropriate category. The classical rule for hypothesis testing when we set a threshold for each of the hypotheses could not be used. The hypothesis testing algorithm is based on the winning score rule, where winning score is the log-likelihood that is closest to zero. We denote the winning hypothesis with H_{WIN}. Hence, the hypothesis testing procedure is as follows:

$$H_{WIN} = \begin{cases} H_1 & \text{if } loglik_1 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_2 & \text{if } loglik_2 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_3 & \text{if } loglik_3 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_4 & \text{if } loglik_4 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \end{cases} \tag{4.6}$$

The first hypothesis, H_0 corresponds to normal behavior and that hypothesis is not used in the classification algorithm. H_0 is used in detection algorithm using the criterion that is presented in figure 4.2.

As seen from the figure, the detection algorithm loads every sequence i, where $i = \{1, \ldots, N\}$ and N is the total number of sequences of length 100 in the given BSM sequence. Then it tests whether the sequence has either shell execution or oversized argument. In either case, the sequence is classified as anomalous. Otherwise the sequence is normal. When all N sequences are tested, the algorithm finishes and proceeds to the classification phase, using only sequences extracted in this phase. The testing in this phase is performed between hypothesis H_0,

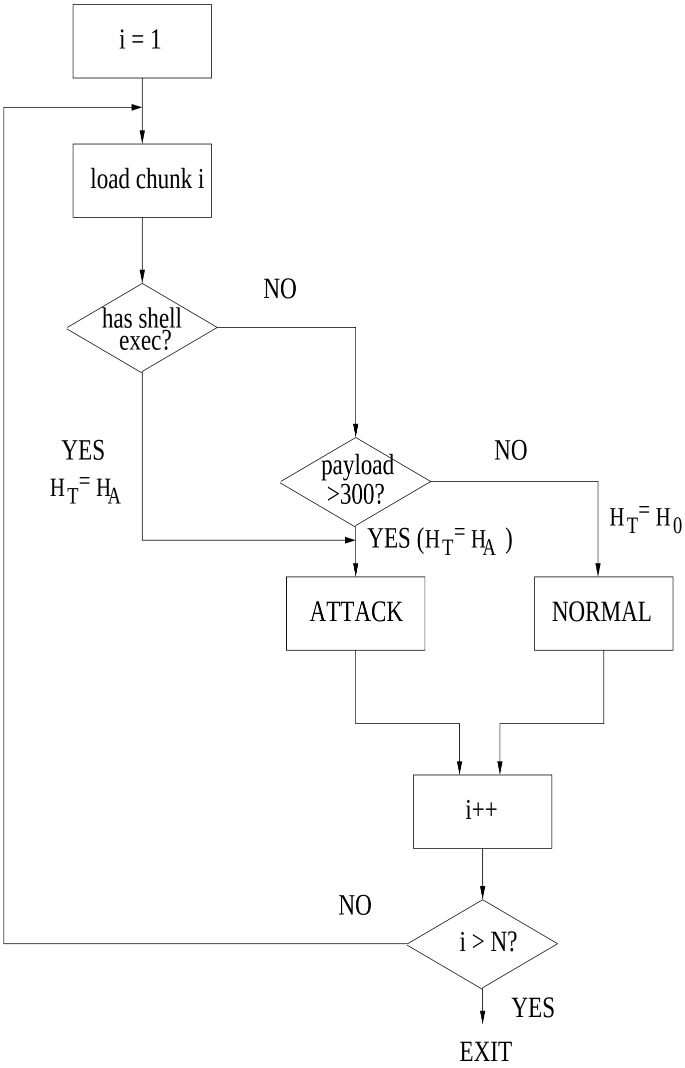


Figure 4.2: Detection of abnormal sequences.

that corresponds to normal sequence and H_A that corresponds to anomalous sequence. Hence, if the sequence contains either shell execution or has payload greater than 300, the winning hypothesis is H_A and the sequence is processed to the classification step. Otherwise, the winning hypothesis is H_0 and the sequence is declared to be normal and is not considered in further steps.

4.3.5 Classification

This section tests on hypotheses H_i, $i = \{1, 2, 3, 4\}$, that correspond to different types of attacks. In case of 4 buffer overflows the hypotheses are: H_1 (ffbconfig), H_2 (fdformat), H_3 (eject) and H_4 (ps). The algorithm is depicted on figure (4.3) and explained in detail in the following sections. However, due to limitations of the data set, that are going to be explained in later sections, the maximum number of hypotheses was 3, corresponding to states with no attack, `ffbconfig` and `eject` attack in one case and no attack, `ffbconfig` and `format` in another case. For theoretical purposes we also considered testing of all hypotheses on one data set that contained only 2 out of 4 attacks.

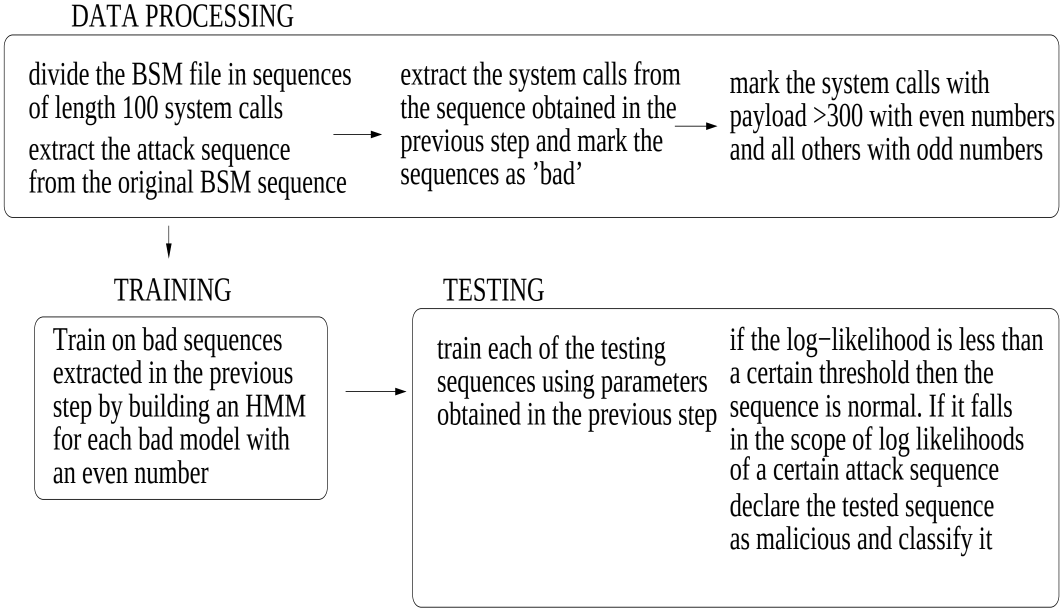

DATA PROCESSING
divide the BSM file in sequences of length 100 system calls
extract the attack sequence from the original BSM sequence
extract the system calls from the sequence obtained in the previous step and mark the sequences as 'bad'
mark the system calls with payload >300 with even numbers and all others with odd numbers
TRAINING
Train on bad sequences extracted in the previous step by building an HMM for each bad model with an even number
TESTING
train each of the testing sequences using parameters obtained in the previous step
if the log−likelihood is less than a certain threshold then the sequence is normal. If it falls in the scope of log likelihoods of a certain attack sequence declare the tested sequence as malicious and classify it

Figure 4.3: Detection and classification system scheme.

Hypothesis testing was preformed on BSM chunks that contained either shell execution or large payload. The sequences were loaded into a column matrix which was used for hypothesis testing. We created a number of HMMs fitted to `ffbconfig, format, eject` and `ps` testing sequences. Parameters obtained during training (π_i, A_i and B_i) were used as input parameters for testing. We calculated the log-likelihood for each of testing sequences using each of the models obtained by training as input parameters. If the tested sequence does not fit the malicious model, the log likelihood ratio converges to $-\infty$ after the first iteration. Each sequence that had log likelihood greater than -400 was classified as corresponding to an attack. Otherwise it was declared to be a false alarm. The threshold of -400 was chosen based on the observation that the minimum log-likelihood observed on training sequences was around -300. Any other small negative value could have been used.

Due to the fact that most of buffer overflows have almost identical structure, the only distinguishing characteristic among those attacks is usually the program that is executed during the attack. In most cases of testing an additional criterion had to be used. If a sequence is classified as both attack 1 and attack 2, two algorithms can be applied to avoid misclassification among the attacks.

4.4 Algorithm 1

This algorithm uses the property that the more similar two models are, the log-likelihood ratio is closer to zero. Hence, if a sequence is classified as both attack 1 and attack 2, the testing algorithm compares the log-likelihood of the sequence when it is classified as attack 1 to the log-likelihood when it is classified as attack 2. The attack with larger log likelihood wins and the sequence is classified accordingly. This algorithm performed correct classification in the majority of cases.

4.5 Algorithm 2

This algorithm is based on the fact that each tested sequence is named according to the line where its system call is placed in the original BSM file. The same program applied in the step 2 of training can be applied for testing since its only role is to determine the line numbers where each of the programs of interest is executed. Simply, if sequence W6Thursday.bsm.split.1600.txt is classified as both attack 1 (attack on program 1) and attack 2 (attack on program 2) and our program determines that program 1 is being executed at line 1600, this sequence is classified as attack 1. When we used this criterion there was no misclassification among the attacks. However, this particular algorithm cannot be used for detection of any attacks other than buffer overflows. For example, if we need to classify between a buffer overflow on `eject` program and some other type of attack on `eject` program this algorithm will not be of any use. But if we determine by the structure of the sequence that the attack is buffer overflow, this algorithm will have 100% classification rate.

Chapter 5

Results

All test performed on data sets include hypothesis testing. Only testing among `ffbconfig` and `eject` attacks and `ffbconfig` and `format` attacks is performed using 3 hypotheses: H_0 for normal, H_1 for `ffbconfig` and H_2 for `fdformat` and H_3 for `eject`. In the first case testing was performed on the actual testing data from Week 6 and in the second case testing was performed on Week 5 Monday due to the lack of `fdformat` attacks in testing data. Week 6 Thursday contained 3 `ffbconfig` and 12 `eject` attacks. Week 5 Monday contained only one `ffbconfig` and one `fdformat` attack. Because of that, we performed training and testing on all possible combination of sequences and the results obtained were almost identical. Hypothesis testing using 5 hypotheses was not possible due to the fact that the testing data set does not contain any `fdformat` or `ps` attacks. For testing those attacks we trained each of those attacks on data from 2 days and tested it on data from the third day. For `ps` attack we used 1999 data.

5.1 Detection of ffbconfig and eject attacks

Due to already presented difficulties, the hypothesis testing was performed using 3 hypotheses. Hypothesis H_0 corresponds to normal sequences, H_1 to `ffbconfig` attack and H_3 to `eject` attack. Testing was performed only on those sequences that had shell execution or even number (the sequences that were extracted in Step 1 of the algorithm). The results obtained in testing phase are first presented in the form of graphs and tables and then discussed.

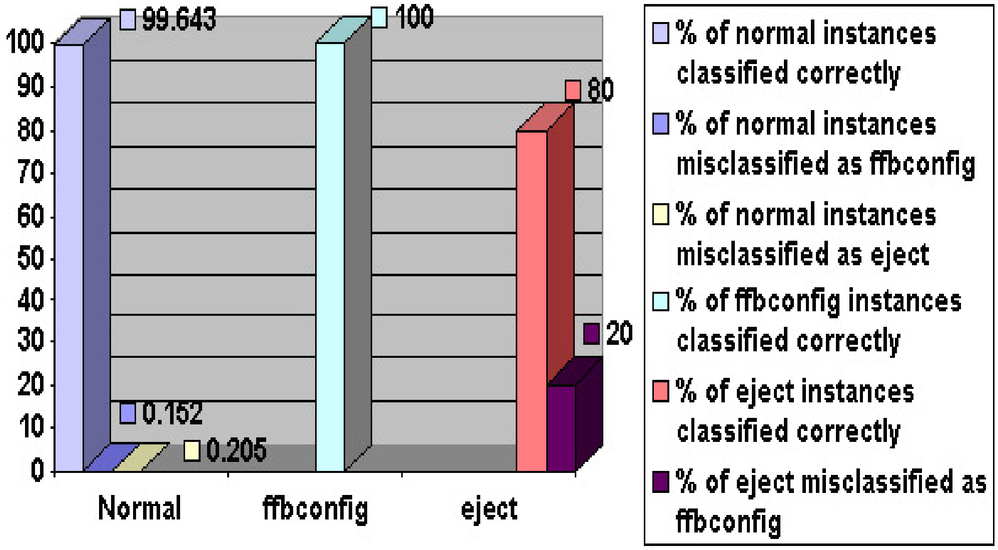


Figure 5.1: Ffbconfig and eject tested on Week 6 Thursday.

Input parameters for tested sequences were HMM parameters obtained during the training phase. No eject attacks were classified as normal and no ffbconfig attacks were classified as normal. The only misclassification that happened was that some sequences were classified as both ffbconfig and eject attacks. There was a total of 9251 sequences. 9233 were normal, 6 sequences characterized ffbconfig attack and 12 sequences characterized eject attack. When Algorithm 1 was applied to the sequences it led to the misclassification rate of 20% for eject attacks (8 out of 10 attacks were classified correctly) and 0% misclassification rate for ffbconfig. When Algorithm 2 was applied it led to misclassification rate of 0%. There were 0.368% of false positives and no false negatives. Hence, the only imperfection of the algorithm reflects in misclassification among different types of attacks, not among normal and anomalous. The results are presented on figure 5.1 and tables 5.1 and 5.2.

	Normal	Anomalous
Normal	99.643%	0.357%
Anomalous	0%	100%

Table 5.1: Confusion matrix for the case of testing between 3 hypotheses: normal, ffbconfig and eject

	Normal	Ffbconfig	Eject
Normal	9200	14	19
Ffbconfig	0	6	0
Eject	0	3	9

Table 5.2: Detection and misclassification rates for the case of testing between normal, ffbconfig and eject

5.2 Detection of ffbconfig and format attacks

Since there are no instances of format attack in weeks 6 and 7, we had to use two weeks from weeks 1-5 for training and one for testing. Applying the same procedure as in the previous case, the false alarm rate was 0.3% and misclassification rate was 0% using either of criterions presented in the previous section.

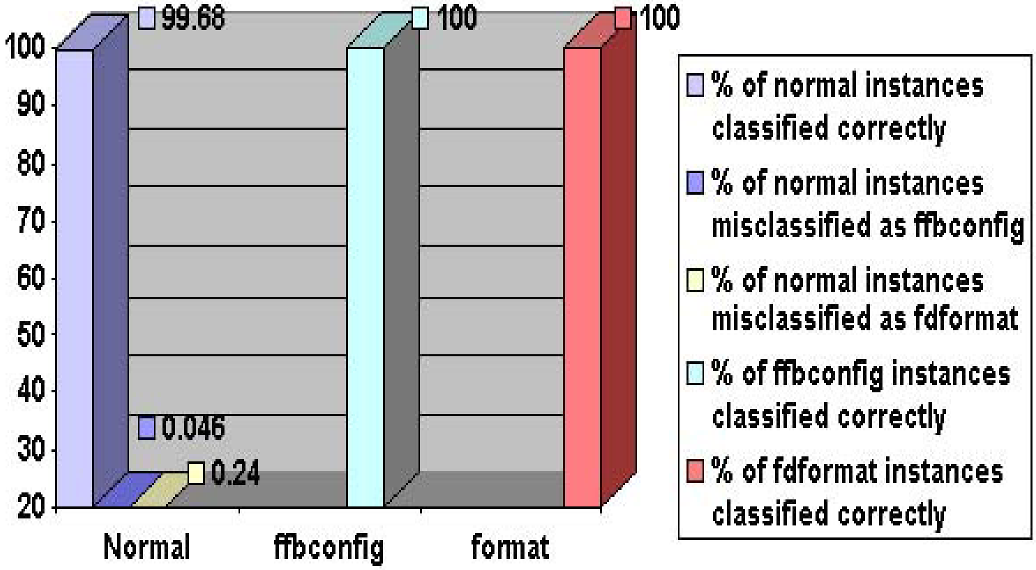


Figure 5.2: Ffbconfig and format detection on Week 5 Monday.

False positive rate was 0.319% and there were no false negatives. No format or ffbconfig attacks were misclassified. The same results were obtained using both algorithms.

	Normal	Anomalous
Normal	99.68%	0.3198%
Anomalous	0%	100%

Table 5.3: Confusion matrix in case of multiple hypothesis testing

	Normal	Ffbconfig	Format
Normal	99.68%	0.046%	0.24%
Ffbconfig	0	100%	0
Format	0	0	100%

Table 5.4: Detection and misclassification rates in case of detection and classification of multiple types of attacks

5.3 Detection and classification of ps attacks

Due to the fact that ps attack appears only in 1999 data, we could not test detection of ps attacks using any other hypotheses except H_0 (normal) and H_4 (ps). The false alarm rate was 0.3% and there was no misclassification, all instances of ps attacks were detected.

	Normal	Ps
Normal	99.53%	0.47%
Ps	0%	100%

Table 5.5: Confusion matrix for detection of Ps ata7k

There were 0.47% false positives and no false negatives. The results are presented on figure 5.3 and in table 5.5.

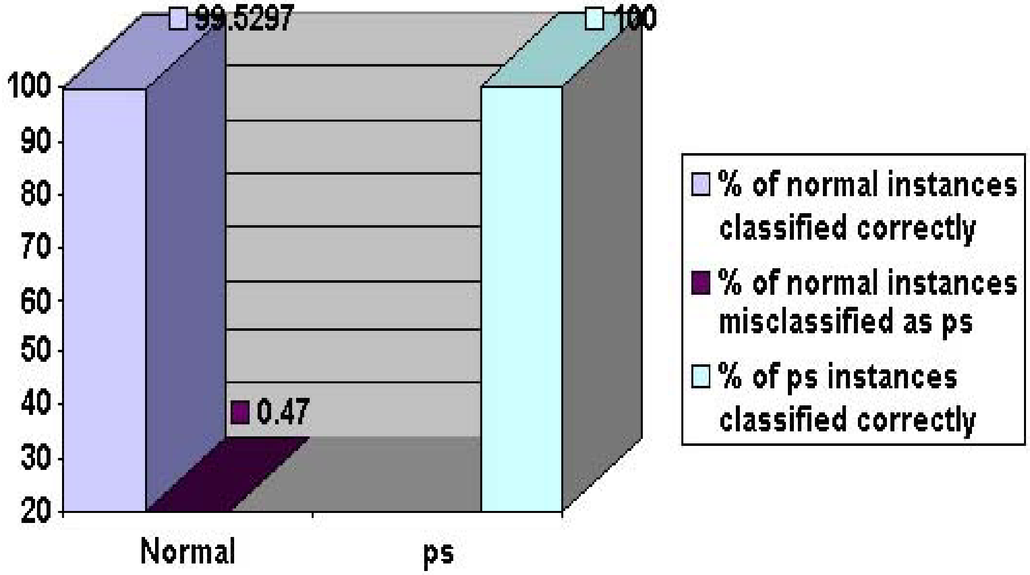


Figure 5.3: Ps detection.

5.4 Detection of eject attacks

The sequences were trained on three weeks of data and tested on the fourth one. Like in the previous cases all instances of attack were classified as anomalous and very low percentage of normal instances were misclassified. The graphical and tabular representations are presented in figure 5.4 and table 5.6.

	Normal	Eject
Normal	99.577%	0.433%
Eject	0%	100%

Table 5.6: Confusion matrix for detection of Eject attack

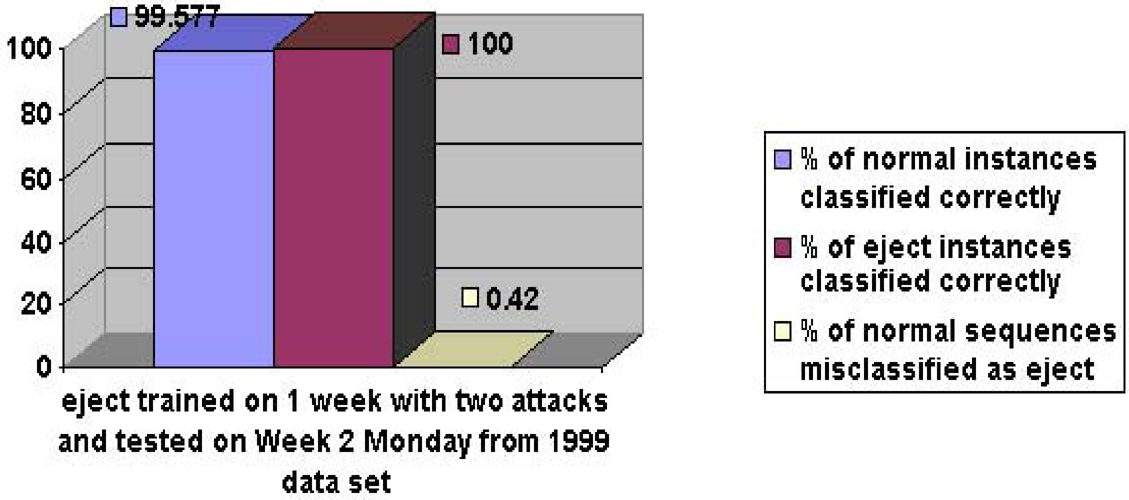


Figure 5.4: Detection of eject attack.

5.5 Detection of fdformat attacks

Fdformat training was performed on data sets from two weeks and tested on data set from one week, that contained one attack. All possible combinations of training and testing data were used and the results were almost identical. The false alarm rate was low and there were no false negatives, as in the previous cases. The results are presented in figure 5.5 and table 5.7. As seen in the table, the confusion matrix is almost symmetric.

	Normal	Fdformat
Normal	99.685%	0.0.315%
Fdformat	0%	100%

Table 5.7: Confusion matrix for detection of Fdformat attack

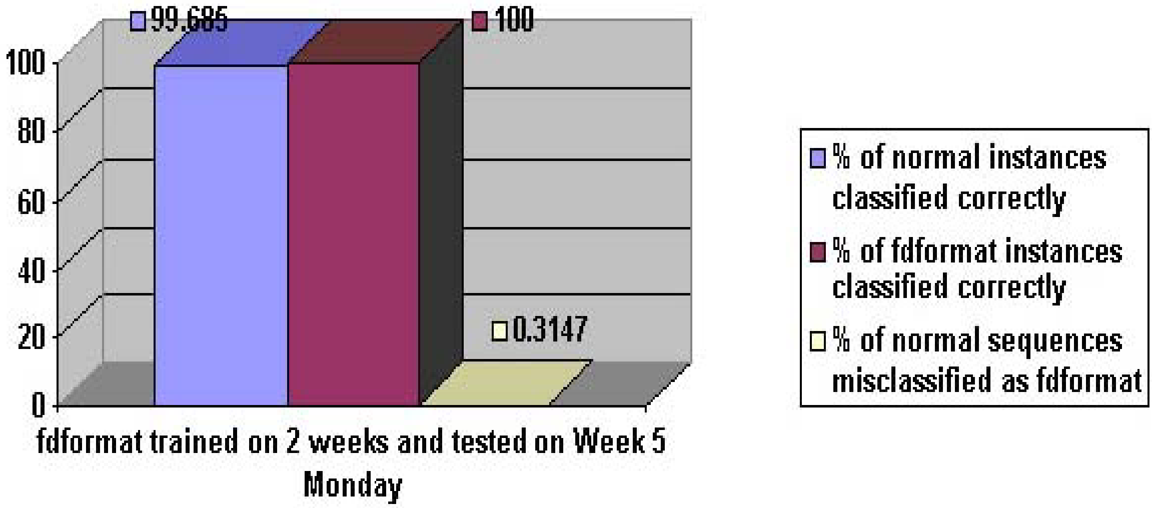


Figure 5.5: Detection of fdformat attack.

5.6 Detection of ffbconfig attacks

Finally the results for detection of ffbconfig attacks are presented in figure 5.6 and the confusion matrix is presented in table 5.8. As we can see, the confusion matrix is again almost symmetric, the false alarm rate is very low and there are no false negatives.

	Normal	Ffbconfig
Normal	99.577%	0.433%
Ffbconfig	0%	100%

Table 5.8: Confusion matrix for Ffbconfig attack

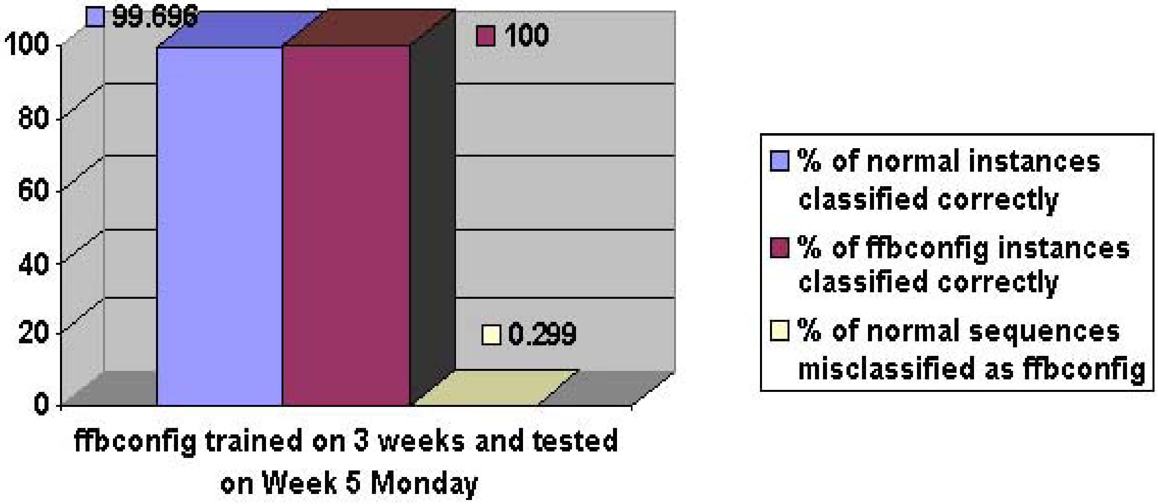


Figure 5.6: Detection rates for ffbconfig attack.

Chapter 6

Conclusions and Contributions

This thesis demonstrated that it is possible to model network attacks with models that consist of small finite number of states. These models can be used for both detection and classification of attacks. The method presented in this thesis was applied and tested only to attacks exploiting buffer overflows due to the non availability of data for testing it on other attacks, like `ftp-write` or race condition attacks. We developed models for other attacks and we believe that they are applicable for detection and classification of other attacks, not only buffer overflows. The algorithm can also be applied for detecting race condition attacks since they add additional loops in program behavior, that do not exist in normal behavior. The method presented in this thesis can be used in combination with static analysis tools to achieve even higher detection rate and classify the attacks that the static analysis tools may have missed.

The algorithm presented in the previous chapter is motivated by the attack models developed in Chapter 3 of this thesis. Due to already mentioned reasons the results were presented only for buffer overflow attacks. Since we needed uniform representation of all attacks with the least amount of additional rules, only key features of those models were selected: large header, execution of a specific program (ffbconfig, eject, etc.) and shell execution. Other specific features of attacks could be used, like large amount of `stat` commands in eject attacks. The feature of all training and testing data sets was that each attack was characterized with `ksh` execution and those were the only appearances of `ksh` in given BSM sequences. However, it is uncertain if the `ksh` execution is a specific characteristic of the MIT Lincoln Labs data set (actually characteristic of machines that were used for data generation, in our examples machine `pascal`) or it is a characteristic of all buffer overflow attacks. There are three different rules for detection of anomalous sequences that can be applied.

1. If there is a `ksh` execution in the sequence classify that specific sequence as anomalous. Otherwise classify it as normal.

2. If there is a sequence that has `execve` system call with large payload (even number after Step 1 of my algorithm) followed by a sequence with `ksh` execution classify either of the sequences as an attack. In this rule, "followed by" does not necessarily mean "immediately followed by". The sequence with `ksh` execution can be separated from the sequence with large payload with several interleaved sequences.

3. Mark all sequences with shell execution (ksh, sh, bsh etc.) as potentially dangerous. Build in the information about payload in B matrix of HMM. Test only on sequences with shell execution and large payload.

The first two algorithms have detection rate of 100%. However, as mentioned before those rules are limited only to a specific data set. The second algorithm was proved to be efficient on this particular year from MIT Lincoln Labs Data set. This method may not be able to detect many of the attacks in case when globbing, described in Chapter 3, is performed since the program will observe only unrelated symbols, but the operating system is going to interpret them, as `ksh` execution, for example. Hence, the attack will be misclassified as a normal sequence and this method may possibly lead to a certain number of false negatives. The third algorithm has very high detection rate, although not 100% as the previous ones. The obvious advantage of this algorithm is that it does not depend on any feature of this specific data set. It is based on general features of buffer overflow attacks: shell execution (any shell, not only `ksh`) and large payload. This method may also miss some shell executions for the same reason as algorithm 2, but in this case it will still have a sequence with oversized argument that can be recognized by trained HMMs. This algorithm leads to around 0.1% to 0.5% of false negatives. Attack detection in this thesis is defined as detection of at least one sequence of that attack. If sequence number 1000 is classified as ffbconfig attack, then automatically the whole program with the process ID that corresponds to sequence 1000 is classified as anomalous ffbconfig program.

The classification rate in most cases is 100% when method 1 described in the previous section is used (observing values of log-likelihoods) and is always 100% when method 2 is used. This justifies the approach taken in this thesis and proves that the behavior of attacks can be completely captured by HMMs.

The contribution of this thesis is reflected in the area of classification. There are a lot of publications that use different models for detection between normal and anomalous. None of them puts emphasis on automatic classification of attacks. Some of the publications use HMMs for detection of attacks but there are no specified methods or algorithms for HMM application or justification for using HMMs for detection. All of those approaches used training on normal sequences which caused high complexity of the algorithm. This thesis uses an approach that is based on training on *anomalous* sequences. Hence, each attack can be represented with 3-4 HMM models. Training is performed very efficiently since it is not performed on the BSM file with hundreds of thousands of system calls, but on a couple of files, each of length 100 system calls. Instead of having either a large number of models for normal sequences or high training time, we have an efficient algorithm that performs testing using only 3-4 models for each attack and testing is performed on around 400 to 700 sequences, which represents around 2-4% of the total number of sequences of the initial data set. The strength of this approach is that it chooses only potential attacks in both the training and testing sets.

This approach was applied and tested in the course of work reported here only for detecting and classifying of buffer overflow attacks. This approach can also be used for detecting race conditions, since race condition attacks are characterized with large number of executions of a certain command, for example deletion of `/tmp` directory. This will result in a specific construction of abnormal HMM which will significantly differ from normal ones. However, this method cannot be applied for detection of race condition attacks that are performed over a long period of time. The strength of this approach is that the attacker cannot change the behavior of normal over time by slowly adding more and more abnormal sequences since we are using anomalous sequences for training.

6.1 Future work

We plan to apply the same (or similar) algorithm to other types of attacks, not only buffer overflows since each attack has some characteristic sequences. The model for `ftp-write` already exists and we plan to test it for detection of race condition attacks. The thesis demonstrated that the method is efficient for both detection and classification. The normal/abnormal detection level is going to perform on the same high level as when applied on buffer overflows. The misclassification of attacks happened mostly in weeks that have both User to Root and Remote to Local attacks. The weeks that had only DoS attacks and buffer overflows had very low false alarm rate and no misclassification. Therefore, we believe that, if rules for detection of other attacks are added, the misclassification rate will be reduced even more. Not all of those attacks can be detected with static checking tools. The most convenient attacks for detection with static analysis are buffer overflows and most of static analysis tools are targeted towards detection of buffer overflows. We can also use the models developed in Chapter 3 for classification between different types of attacks, specifically, to classify whether the attack is buffer overflow or some other type of attack. The main problem is lack of data since we cannot claim that the algorithm works if training is performed on one sequence and testing is performed on another sequence from the same week or by using an attack performed by the same intruder. That is the main reason why this extension was not applied during the work reported here . The attacks that had most instances were buffer overflows and that is the reason why we applied and tested our method on such attacks. This algorithm is used for detection of already known attacks. However, it can always classify the sequence as normal or anomalous. If the sequence is classified as anomalous and it does not fit any of the existing models, it can be named as *unknown* attack and passed on to another system that will determine if the attack is an already known attack that has been altered or a completely new attack.

Bibliography

[1] Stuart McClure et. al "Hacking Exposed: Network Security Secrets & Solutions", Third Edition, McGraw Hill, 2001.

[2] Richard P. Lippmann, Robert K. Cunningham "Guide to Creating Stealthy Attacks for the 1999 DARPA Off-Line Intrusion Detection Evaluation", MIT Lincoln Laboratory. 24. April 2000.

[3] Kriss Kendal "A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems", M.Eng. Thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.

[4] Richard P. Lippmann et. al. "Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation", Recent Advances in Intrusion Detection 2000: 162-182

[5] A. K. Ghosh and A. Schwartzbard "A study Using Neural Networks for Anomaly and Misuse Detection", in Proceedings of the USENIX Security Simposium, August 23-26, 1999, Washington, D.C

[6] Deborah Russel and G.T. Gangemi Sr. "Computer Security Basics", O'Reilly & Associates, Inc., 1991.

[7] S. Jajodia, D. Barbara, B. Speegle and N. Wu, Audit Data Analysis and Mining (ADAM), project described in http://www.isse.gmu.edu/˜dbarbara/adam.html, April 2000

[8] P. Neumann and P. Porras, "Experience with EMERALD to DATE", in Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, April 1999, 73-80

[9] A. Schwartzbard and A. K. Ghosh,"A study in the Feasibility of Performing Host-based Anomaly Detection on Windows NT", in Proceedings of the 2nd Recent Advances in Intrusion Detection (RAID 1999) Workshop, West Lafayette, IN, September 7-9, 1999.

[10] R. Sekar and P. Uppuluri, "Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications", in Proceedings 8th Usenix Security Symposium, Washington DC, Aug. 1999.

[11] M. Tyson, P. Berry, N. Williams, D. Moran, D. Blei, "DERBI: Diagnosis, Explanation and Recovery from computer Break−Ins", project described in http://www.ai.sri.com/˜derbi, April 2000.

[12] G. Vigna, S. T. Eckmann and R. A. Kemmerer, "The STAT Tool Suit", in Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX), IEEE Press, January 2000.

[13] Stephen E. Smaha "Haystack: An Intrusion Detection System", Proc. of the IEEE 4th Aerospace Computer Security Applications Conference, Orlando, FL, Dec. 1988, 37 - 44

[14] T. Lane, C. Brodley, “An application of machine learning to anomaly detection”, Proc. 20th NIST-NCSC National Information Systems Security Conference, 1997.

[15] H. Teng, K. Chen, S. Lu, “Security Audit Trail Analysis Using Inductively Generated Predictive Rules”, Proceedings of the sixth conference on Artificial intelligence applications January 1990.

[16] Garth Barbour “Program Modeling: A Machine Learning Approach To Intrusion Detection”, PhD Thesis, University of Marryland at College Park, 2002.

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji and T. A. Longstaff “A sense of self for unix processes”. In Proceedings of the 1996 IEEE Symposium on security and Privacy, pages 120-128, Los Alamitos, CA, 1996. IEEE Computer Society Press

[18] S. A. Hofmeyr, S. Forrest, A. Somayaji “Intrusion detection using sequences of system calls” Journal of Computer Security Vol. 6, pp. 151-180 (1998).

[19] C. Warrender, S. Forrest, B. Pearlmutter “Detecting Intrusions Using System Calls: Alternative Data Models”, 1999 IEEE Symposium on Security and Privacy, May 9-12, 1999.

[20] C. Marceau, “Characterizing the Behavior of a Program Using Multiple-Length N-grams”, Proceedings of the New Security Paradigms Workshop 2000, Cork, Ireland, Sept. 19-21, 2000.

[21] A. Wespi, M. Dacier, H. Debar ”An Intrusion-Detection System Based on the Teiresias Pattern Discovery Algorithm ”, Technical Report RZ3103, Zurich Research Laboratory, IBM Research Division, 1999.

[22] I. Rigoutsos, A. Floratos ”Combinatorial Pattern Discovery in Biological Sequences: the TEIRESIAS Algorithm”, Bioinformatics, Vol. 14, No. 1, 1998,pp. 55-67.

[23] A. Wespi, M. Dacier, H. Debar “Intrusion Detection Using Variable-Length Audit Trail Patterns”, Recent Advances in Intrusion Detection - Lecture Notes in Computer Science ed. by H. Debar, L. M, S.F. Wu. , Berlin, Springer-Verlag, vol.1907, p.110-29 in 2000

[24] A. P. Kosoresow, S. A. Hofmeyr, “Intrusion Detection via System Call Traces”, IEEE Software, vol. 14, pp. 24-42, 1997.

[25] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, “A Fast Automation-Based Method for Detecting Anomalous Program Behaviors”, IEEE Symposium on Security and Privacy, Oakland, California,May 2001.

[26] A. K. Ghosh, A. Schwartzbard, M. Schatz, “Learning Program Behavior Profiles for Intrusion Detection”, Proceedings of the Workshop on Intrusion Detection and Network monitoring, USENIX 1999.

[27] C. Ko, G. Fink, K. Levitt, “ Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring”, Computer Security Application Conference, 1994.

[28] R. Sekar, P. Uppuluri, “Experiences with Specification-Based Intrusion Detection”, RAID October 2001.

[29] N. Nuansri, S. Singh, T. S. Dillon, “A Process State-Transition Analysis and its Application to Intrusion Detection”, 15th Annual Computer Security Applications Conference December 06 - 10, 1999 Phoenix, Arizona

[30] D. Barbara, *et.al.*, “ADAM: Detecting Intrusions by Data Mining”, Proceedings of the 2001 IEEE Workshop on Information Assurance and Security, June 2001.

[31] P. A. Porras and P. G. Neumann, “EMERALD: Event monitoring Enabling Responses to Anomalous Live Disturbances”, In Proceedings of the National Information Systems Security Conference, 1997, pp. 353-365.

[32] J. Habra, B. Charlier, A. Mounji, I. Mathieu, “ASAX: Software architecture and rule-based language for universal audit trail analysis”, Second European Symposium on Research in Computer Security (ESORICS 92).

[33] G. Vigna, S. T. Eckmann, R. A. Kemmerer, “The STAT Tool Suite”, in Proceedings of DISCEX 2000, Hilton Head, South Carolina, January 2000, IEEE Press.

[34] K. Ilgun, “USTAT: A real-time intrusion detection system for UNIX”, in Proceedings of the 1993 IEEE Symposium on Security and Privacy, Oakland, CA, May 1993.

[35] S. T. Eckmann, G. Vigna, R. A. Kemmerer, “STATL: An Attack Language for State-based Intrusion Detection”, in Proceedings of the ACM WOokshop in Intrusion Detection, Athens, Greece, November 2000.

[36] O. Sheyner, J. Haines, S. Jha, R. Lippmann, J. M. Wing, “Automated Generation and Analysis of Attack Graphs”, 2002 IEEE Symposium on Security and Privacy May 12 - 15, 2002 Berkeley, California, p. 273

[37] NuSMV, “NuSMV: A new Symbolic Model Checker”, `http://afrodite.itc.it:1024/ nusmv`

[38] Aleph One, “Smashing the stack for fun and profit”, Phrack magazine, volume 7, issue 49.

[39] Rabiner, L. R. *A tutorial on Hidden Markov Models and selected applications in speech recognition*, Proc. IEEE, 77(2), 257-286.

[40] Andrew P. Moore, Robert J. Ellison and Richard C. Linger, “Attack Modeling for Information Security and Survivability”, Technical Note, CMU/SEI-2001-TN-001

[41] Bruce Schneier, “Attacks Trees”, Dr. Dobb’s Journal, December 1999.

[42] David Evans, David Larochelle ”Improving Security Using Extensible Lighweight Static Analysis”, IEEE Software, January/February 2002.

[43] David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken ”A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”, The 2000 Network and Distributed Systems Security Conference. San Diego, CA ,February 2000.

[44] David Wagner, Hao Chen ”MOPS: an Infrastructure for Examining Security Properties of Software”, 9th ACM Conference on computer and communications security, Washington DC, November 2002.