T492

UNIVERSITY LIBRARY * COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY * COCHIN 682 022

AN ALTERNATIVE APPROACH TO COMPUTER SYSTEM SECURITY MONITORING AND ENHANCEMENT THROUGH SYSTEM CALL SEQUENCE ANALYSIS

A THESIS

Submitted by

SUREKHA MARIAM VARGHESE

for the award of the degree of

DOCTOR OF PHILOSOPHY

UNDER THE FACULTY OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
KOCHI, KERALA 682 022

MARCH 2008

CERTIFICATE

This is to certify that the thesis titled ***'An Alternative Approach To Computer System Security Monitoring And Enhancement Through System Call Sequence Analysis'***, is a report of the original work carried out by **Ms. Surekha Mariam Varghese**, under my supervision and guidance in the Department of Computer Science, Cochin University of Science and Technology. The results enclosed in this thesis or part of it have not been presented for any other degree.

[illegible]

Kochi 22
11-03-08.

Dr. K. Poulose Jacob,
(Supervising Teacher)
Professor & Head,
Department of Computer Science,
CUSAT, Kochi.

DECLARATION

I here by declare that this thesis is a report of the original work done by me under the supervision of Dr. K. Poulose Jacob in the Department of Computer Science, Cochin University of Science and Technology, and that no part thereof has been presented for any other degree.

[illegible]

Kochi 22

11-03-08.

Surekha Mariam Varghese

ACKNOWLEDGEMENTS

First of all I am grateful to the **GOD almighty**, for guiding my life and for all the blessings showered upon me for the successful completion of this work, my long cherished dream.

I wish to place on record my profuse sense of gratitude and sincere thanks to

Dr. K. Poulose Jacob, Director, School of Computer Science Studies for his guidance, encouragement and support throughout the period of research, without which successful completion of the program would not have been possible. His caring nature and concern for the students is greatly appreciated.

Staff of Cochin University of Science and Technology for all the help and timely advices.

My husband **Dr. Laju Kottalil** for his love, constant support and encouragement. My sons **Sachin Basil John** and **Sharath Basil Kurian** deserve special thanks for their love, affection and for sparing me without any complaint.

My parents **Prof. T. K. Varughese** and **Mrs. Ammini Varughese** and, parents-in-law **Prof. K. A. Johnie** and **Prof. (Mrs.) Leela John,** for their affection, constant encouragement and support throughout my life. They were the constant source of inspiration for completing this research program.

Authorities of **M.A.College of Engineering Kothamangalam**, Kerala for the constant encouragement.

My colleagues at Department of Computer Science and Engineering **Joby George, Jisha P. Abraham, Aby Abahai T, Linda Sara Mathew, Eldo P. Elias, Aswathi P. Sivadas, Jomon Mathew, Jiso George,** and **Anish P.K.** for their sincere help, valuable suggestions, discussions, and for the generosity to relieve me from various academic responsibilities during the final stages of the research work.

My friend **Sheena Mathew** for her moral support, help, and academic discussions.

All relatives and well wishers for their support in critical hours in completing this program.

Surekha Mariam Varghese

ABSTRACT

Keywords: *Buffer Overflow, Anomaly Detection, Intrusion, Security.*

Modern computer systems are plagued with stability and security problems: applications lose data, web servers are hacked, and systems crash under heavy load. Many of these problems or anomalies arise from rare program behavior caused by attacks or errors. A substantial percentage of the web-based attacks are due to buffer overflows. Many methods have been devised to detect and prevent anomalous situations that arise from buffer overflows. The current state-of-art of anomaly detection systems is relatively primitive and mainly depend on static code checking to take care of buffer overflow attacks. For protection, Stack Guards and Heap Guards are also used in wide varieties.

This dissertation proposes an anomaly detection system, based on frequencies of system calls in the system call trace. System call traces represented as frequency sequences are profiled using sequence sets. A sequence set is identified by the starting sequence and frequencies of specific system calls. The deviations of the current input sequence from the corresponding normal profile in the frequency pattern of system calls is computed and expressed as an anomaly score. A simple Bayesian model is used for an accurate detection.

Experimental results are reported which show that frequency of system calls represented using sequence sets, captures the normal behavior of programs under normal conditions of usage. This captured behavior allows the system to detect anomalies with a low rate of false positives. Data are presented which show that Bayesian Network on frequency variations responds effectively to induced buffer overflows. It can also help administrators to detect deviations in program flow introduced due to errors.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

All advanced forms of life are gifted with self-awareness. Because of this consciousness, living systems are able to detect and respond to changes in their internal state. In contrast, computer systems normally fail to detect and respond to changes in their behavior. In the current scenario most of the operating systems do not have sufficient mechanisms to thoroughly profile or monitor the system behavior.

As a result many a times, systems deviate from the normal behavior and lead to unauthorized access and other security violations. Unusual program behavior often leads to data corruption, program crashes, and denial of services. Despite these problems, current computer systems have no effective general-purpose mechanism for detecting and responding to such anomalies.

This dissertation is an attempt towards overcoming this shortcoming. It is based on the idea that computer operating systems can be augmented with mechanisms to improve security, without much compromise on system performance. In the ensuing chapters, a prototype system, based on system call frequency representation, to profile processes and to detect anomalies during process executions, is proposed and evaluated. It can detect changes in program behavior and is particularly good at responding to security violations caused by buffer overflows.

1.1. MOTIVATION

As our computer systems have become increasingly complex, they have also become more unpredictable and unreliable. Today we routinely run hundreds of programs on any given computer. Many of these executables require tens of megabytes of memory and hundreds of megabytes of disk space. As our systems become faster and larger, programs continue to expand in size and complexity.

Although these programs contain a remarkable amount of functionality, the additional capabilities have necessitated a correspondingly higher cost in reliability and security. New vulnerabilities are found almost every day on most major computer platforms. Even worse, we have all become trained to program quirks and outright crashes. No sooner does one version seem to stabilize, than a new one comes out providing new capabilities and new problems. One of the main drivers of this rise in complexity is the need to be connected, both in local area networks and on the Internet. Web browsers and chat clients continuously communicate with the outside world, peer-to-peer file sharing services turn workstations into servers, and even word-processors have become Internet-aware, able to transfer documents to and from web servers with a mouse click. Thus, the complexity of flexible user-friendly software is compounded by the need to interact with unpredictable outside data sources. While an isolated system might behave consistently over time, a networked system forms a new system every day, as the rest of the Internet changes around it. These two factors, namely complexity and connectivity combine to undermine our trust in computers. Fundamentally, there is too much going on for even the most sophisticated user to keep track of. If we cannot keep track of what our systems are doing, our computers must monitor what they do.

Imbibing self-awareness and consciousness to our system, or attempting to add a little bit of intelligence to the system, is the motivation behind this work.

1.2. PROBLEM FORMULATION

Most computer users would like a system to function consistently over long time, since installation. Given that such consistency cannot always be achieved, it would be nice to know when new circumstances cause a change in system behavior, especially if that behavior could cause problems. Most of the time a program's behavior is confined to the inner circle of normal behavior around codes which are used often. A program wanders out of this region into that of unusual but legal program regions when strange events such as communication failures or invalid input data cause normally unused code to be executed.

If hardware has never executed code incorrectly, and if our programs have perfectly accounted for all possible errors and interactions, then all legal program behavior would be permissible and safe, and no other behavior would be possible. In practice, our programs are far from perfect, and the interactions of these imperfect programs can cause one program to behave like another program, like for example turning a word processor into a file deletion utility. Such unexpected functionality can lead to security violations and data loss.

One way to handle dangerous program behavior is to create detectors to recognize these states explicitly. This approach is used by most misuse intrusion detection systems, usually through the scanning of data streams for attack signatures. Although it can be effective, this strategy has two basic limitations. One is that, generally it is impossible to specify all the ways in which programs can malfunction, and so we are left with systems that must be continually updated as new threats arise. The other is that, a dangerous pattern of activity in one context may be perfectly valid in another; thus, signatures in software products must be created for the lowest common denominator to minimize false alarms.

The new approach, however, has been to recognize the fact that most programs work correctly most of the time, and so normal program behavior is almost always safe program behavior. By learning normal program behavior through observing a system, and by using this knowledge to detect and respond to unusual, potentially dangerous program behavior, we can improve the stability and security of our computer systems.

The fundamental idea of this dissertation is that we can improve the security and integrity of computer systems by ensuring that processes do not deviate from their normal behavior. Practical computer users are often found to avoid program upgrades, precisely to avoid disturbing their stable computational environment. The idea is to evolve a system that can detect and respond to harmful system changes, whether they are caused by a configuration error, misuse of a service, or an outright attack.

1.3. ANOMALY DETECTION

Normally programs make well-defined sequences of system calls, perform predefined steps and invoke functions in a particular fashion. Any deviation from this normal behavior or flow of control can be called an anomaly. All anomalies are not very harmful. But many anomalies lead to unsafe situations where there is a potential possibility of a deliberate unauthorized attempt to access information, manipulate information, or render a system unreliable or unusable.

The most commonly exploited vulnerability in general-purpose operating systems is the buffer overflow. It is encountered due to insufficient bounds checking on arguments that are supplied by users and occurs whenever a request for a buffer access crosses the array / buffer boundary that was allocated for it. In Linux almost every Internet application, including Sendmail daemon, Finger daemon, Talk daemon etc., has at least one point from which it can be compromised through a buffer overrun. Many attackers use this method for overwriting data that controls the program execution path and hijack control of the program to execute the attacker's code in place of the process code. So programmers who intend to write codes to be run as the root, or perform inter process communication, or wish to use external files, have to be careful about the buffer overruns.

In an overflow attack, the objective of the attacker is to corrupt information in a carefully designed manner. Commonly they make use of functions that do not check the size of the arguments and pass very large strings as argument to these functions, which either overwrites the function return address or places an executable code in the stack. One method to prevent such attacks is to provide range checking for arrays or buffers used. Owing to the heavy overhead involved, it is usually not preferred. Also for languages like C where the size of arguments are unknown in most cases, range checking is not a good option. The usual method is to go for static code checking to determine risky functions and to substitute them with safer functions. Whenever a new application is developed or a new version is released the same process has to be applied to make it safe. If range checking is applied after analyzing the situation by using a number of parameters, the overhead can be reduced. Also in situations where range checking cannot be directly applied the same parameters can be used to categorize whether the current state is safe or not. Since the verification process is done during runtime the method is quite accurate.

Race condition also leads to anomalies. A race condition is a flaw in a system or process where the output exhibits unexpected critical dependence on the relative timing of events. These are time-of-check-to-time-of-use (TOCTTOU) attacks arising as a result of lack of mutual exclusion. TOCTTOU attacks occur when a system utility (program) performs some important operation at a time t2 after ensuring that some condition C holds at time t1 such that t2 succeeds t1 and at time t2, C does not hold. A malicious program can execute in the interval (t1, t2) even when C does not hold. This can cause the utility to perform an unintended action. Xterm, Binmail, Passwd, Sendmail etc. are vulnerable to race condition based attacks **[Goyal 03]**. In computer memory a race condition may occur if commands to read and write large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read. This is very common in systems, which support Remote Procedure Call. This is especially true in networks with long lag times, because of the high probability of granting multiple remote requests. Occurrence of race condition depends on many factors such as CPU speed, network latency, channel speed, system load at the target station.

Both buffer overflow and race condition may lead to denial of service attacks. CPU, memory and other resources may be used up at a higher rate rendering the system nonfunctional. A steep rise in the utilization rate of the resources beyond the expected level can be predicted as a denial of service attack.

1.4. SYSTEM-CALL MONITORING AND RESPONSE

On UNIX and UNIX-like systems, user programs do not have direct access to hardware resources; instead, one program, called the kernel, runs with full access to the hardware, and regular programs must ask the kernel to perform tasks on their behalf. Running instances of a program are known as processes. Multiple processes can execute the same program; even though they may share code, each process has its own virtual memory area and virtual CPU resources. The kernel shares memory and processor time between processes and ensures that they do not interfere with each other.

When a process wants additional memory, or when it wants to access the network, disk, or other I/O devices, it requests these resources from the kernel through system calls. Such calls normally take the form of a software interrupt instruction that switches the processor into a special supervisor mode and invokes the kernel's system call dispatch routine. If a requested system call is allowed, the kernel performs the requested operation and then returns control either to the requesting process or to another process that is ready to run.

With the exception of system calls, processes are confined to their own address space. If a process is to damage anything outside this space, such as other programs, files, or other networked machines, it must do so via the system call interface.

Unusual system calls indicate that a process is interacting with the kernel in potentially dangerous ways. Interfering with these calls can help prevent damage; therefore, a computational mechanism that monitors and responds to unusual system calls can help maintain the stability and security of a computer system.

1.5. RESEARCH QUESTION

The main research objective of this work is to develop a system that will detect anomalies occurring in the computing system and to explore ways and means to enhance its security. To achieve the above research objective the following research question is formulated:

"How can we enhance the security of the computer system by effectively monitoring anomalies?"

In order to answer the main research question given above, the following sub-questions are to be dealt with:

1. How do anomalies arise and what are the counter measures in practice?
2. Which is the most crucial and unresolved vulnerability and how does it occur?
3. Will buffer overflows change the execution sequence of processes?
4. What are the approaches for intrusion detection? Which one of them is suitable for buffer overflow detection?
5. Is it possible to use system call traces, commonly used in general anomaly detection, to detect anomalies caused specifically by buffer overflows?
6. How can system call traces be collected and represented?
7. What are the criteria for selecting a process for buffer overflow experiments?
8. How can we profile normal behavior of processes and how can we classify input processes as normal and abnormal?
9. What are the parameters for measurement of effective performance of the detection mechanism?
10. How do we develop and evaluate a model based on the concepts?

1.6. RESEARCH METHODOLOGY

Many of the security problems and anomalies (rare program behaviors) are caused by attacks (unauthorized programs). There are many security threats and there are diverse manners in which the attacks can be carried out. . Hence, a thorough literature study was conducted to identify the possible methods of attack and the solutions available to take care of the situations caused by such attacks. A close examination of the existing data revealed that, buffer overflows constitute a substantial portion of the attacks. Hence it was decided to focus on buffer overflow vulnerability.

A detailed study was conducted on intrusion detection to determine the approach to be used for detecting rare program behavior. Intrusion detection techniques are traditionally categorized into two classes: anomaly detection and misuse detection. Anomaly detection is based on the normal behavior of a subject such as a user or a system; any action that significantly deviates from the normal behavior is considered intrusive. Misuse detection catches intrusions in terms of characteristics of known attacks or system vulnerabilities; any action that conforms to the pattern of known attack or vulnerability is considered intrusive.

Buffer overflow is an exploitation technique, and any attack code can be run during exploitation. There is no specific attack signature for buffer overflow. So the misuse detection approach, where, the intrusions are detected by identifying the attack signature becomes complex and not suitable for buffer overflow detection. Hence it was decided to use anomaly detection approach, and to profile the normal behavior of processes, so as to detect deviations from the normal as anomalous.

Further survey of literature was focused on how buffer overflows are utilized by attackers to gain unauthorized access. Initial experiments were conducted in RedHat Linux to exploit simple processes using buffer overflows. Buffer overflows were created by passing very large strings containing intrusion code. It was observed that buffer overflows often change the execution sequence of processes.

Many researchers have used system call traces generated by processes to profile process executions. The next set of experiments verified whether buffer overflows could be detected by tracing system calls generated. System call traces were collected using the command *strace*. It was clear from the experiments that buffer overflows could be detected by analyzing the system call sequences. The next question was to identify a suitable representation for the system call sequences, which is compact and which would ease the course of detection. The commonly used methods, subsequences and look-ahead pairs, were found to be complex and using much storage space. It was seen that the method using frequency representation of system calls is very compact and simple, though some information (appearing order and parameters) is lost. Further experiments concentrated on frequency representation. The system call sequences were represented using frequencies of around 200 system calls of the system. It was observed that during buffer overflows, frequencies of certain system calls in the frequency sequence get changed. Further experiments were conducted on Sendmail daemon, a commonly used program that is vulnerable to buffer overflows. It was ensured that buffer overflows can be detected by measuring variations in frequency count of system calls of the system. A new approach to represent normal frequency profiles using the concept sequence sets was identified and developed. A simple Bayesian network was used to classify the input profile into normal and abnormal.

A concept prototype was developed and implemented to detect buffer overflows. To ensure the correctness of method, more varieties of traces were needed. To experience the wide varieties of executions and for a standard evaluation, bench marked data sets of Sendmail available at University of New Mexico for buffer overflow was used. Only normal sequences were used for training. Samples were selected using *random()* function from the list of normal sequences. The prototype was tested using random samples from the list of normal and abnormal sequences.

1.7. RESEARCH FINDINGS

The main research findings can be listed as follows.

- Buffer overflows constitute a major portion of attacks.
- Anomaly detection approach is more suitable for detection of buffer overflow attacks.
- Buffer overflow attacks can be detected by analyzing system call traces generated by the process.
- Frequencies of some system calls change due to buffer overflows and frequency representation of system call traces is good enough to detect buffer overflows.
- System call traces represented as frequency sequences can be grouped into sequence sets and frequency profiling using sequence sets captures the normal behavior of processes.

- Bayesian network on frequency variations responding effectively to induced buffer overflows.
- The use of sequence sets allows the system to detect anomalies with a low rate of false positives.
- It can also help administrators to detect deviations in program flow introduced due to errors.

1.8. CONTRIBUTIONS

This work makes several concrete contributions. It monitors the system calls executed by every process on a single host, maintaining separate profiles for every executable that is run. As each process runs, the possible normal sequences of system calls issued by the particular process is collected and stored. Once the possible sequences are collected, the system will be ready to check for anomalies. It supports the idea presented in the past work that system call trace can be used to detect anomalies and ensures that it can be extended to detect novel attacks caused by buffer overflows, Trojan code, and any type of anomaly that causes deviation in the system call sequence. System call monitoring can also detect other problems such as configuration errors, helping administrators to find problems before a loss of service is reported.

1.9. OVERVIEW

The rest of this dissertation is organized as follows. Chapter 2 presents essentials of computer security, intrusion detection and related work in the area. Chapter 3 describes process profiling using system call sequences. Chapter 4 discusses buffer exploitation techniques while Chapter 5 describes the detection model developed and the detection experiments. Chapter 6 presents the performance evaluation of the model. Chapter 7 presents a discussion on the shortcomings. The thesis ends in Chapter 8, where the ideas for future work are described.

CHAPTER 2

COMPUTER SECURITY

With the advent of World Wide Web, Computer Security is receiving much attention. Some significant efforts were taken by developers and entrepreneurs to enhance application level security, which contributed to technological developments in the field of cryptography. Secure protocols such as SSL, SHTTP have brought privacy and authenticity to Web sessions, but they do not secure the computers that are connected to the Internet.

The field of computer security ultimately deals with the problem of unauthorized or dangerous computer access. Covered within this umbrella are issues of data integrity, confidentiality, and system availability. When dealing with these problems, there are three basic approaches one can take:

1. Build systems in such a way that dangerous or unauthorized activities cannot take place.
2. Detect such activities as they happen and ideally stop them before systems are compromised.
3. Detect systems that have been compromised, and determine how much damage has been done.

Clearly the first option is preferable, and before the rise of the Internet this was the main focus of the field.

2.1. OPERATING SYSTEM SECURITY – MAJOR ISSUES

A system is secure only if all the data residing in it and all the resources it manages are secure. The level of protection provided should be in accordance with the criticality of the data and the resources managed by the operating system. Information in a banking system may have great financial value. In a medical system mishandling of information may lead to a life or death problem. A secure operating system can be defined as the resident software controlling hardware and other software functions in an information system that provide a level of protection or security appropriate to the classification, sensitivity, and/or criticality of the data and resources it manages **[Its 01]**.

Sharing and protection are two contradictory goals **[Deitel 00]**. There should be a compromise between these two. It should be ensured that only authorized users use the available resources, in only authorized ways. In a networked environment data has to be protected from unauthorized retrievals, at the client, at the server, and while transmission. The major security aspects at the client and server are resource protection and user authentication. There should be mechanisms in the underlying operating system to protect objects in the network from unauthorized access and to assure the receiver that the message came from the actual source and that it has traveled to him with out being modified.

The best possible method for data security during transmission is the use of encryption. Many public key and secret key algorithms are available for this purpose and currently most of the applications use this technique as the basic security mechanism. In a network scenario the following are the major security flaws that compromise security in the common operating systems.

2.1.1. Inadequate Security Mechanisms:

Most of the current operating systems require some kind of password authentication before logging on to the system. But most of the passwords are common names, phrases or text only words, all of which can easily be guessed. Password protection can also be bypassed by accessing world readable password files, capturing the passwords when it travels through the net (sniffing), by deriving the original password from the encrypted password or hijacking a session after successful login. Dictionary attacks are possible even in cases like Windows NT where only password hash is stored. But if the hash value incorporates "salt" a secret random value such as in UNIX, performing a dictionary attack is not very easy. In a networked environment authentication is even more tedious because the pass phrase has to travel through the net. The current standard protocols do not send password information directly across the network. Instead it uses a challenge response mechanism, where the client sends the stored hash value in response the challenge posed by the server. But the fact that client needs to know only the hash value makes it susceptible to modified client attacks. Now there are single-sign-on products integrating multiple password systems. These authentication systems will help to remove 'clear-text' passwords from the common protocols such as Telnet and FTP [**Visser 97**].

For authorization most of the multi-user operating systems depend on privilege levels and access controls. The concept of privilege levels is applied to the processes in the hardware while that of access controls is applied to the operating system resources. The operating system kernel executes in the highest level, and has full access rights. User processes execute in the lowest ring, and have access only to limit the memory segments.

If a user process violates access restriction or tries to increase its privilege level, then the control is passed to the kernel. The kernel then handles the error, either as a segmentation fault (UNIX) or general protection fault (Windows) **[Mahoney 00]**.

Access control mechanisms decide who can perform which operations and on what objects. There are many access control approaches **[Goscinski 92, Milankovic 00]**. The first is the access matrix model. It provides general representation of access control policies in which the access rights of each subject (user, process) with respect to each object (resource, process) are defined as entries in a matrix. This sparse access matrix is usually implemented using access control lists and capability lists. Though other models based on mandatory access control are available only access control lists and capabilities are used in common operating systems. It is necessary to emphasize that few operating systems so far provide protection mechanisms that prevent unauthorized accessing of system objects.

Inadequate and improper access controls lead to security violation. Some system administrators fail to effectively implement even the available access control mechanisms, by granting excessive privileges to some users. Spoofing attacks, Viruses, Trojans all are results of inadequate access control mechanisms. Some systems allow the concept of trusted hosts. Intruders can impersonate these trusted hosts and get unauthorized access (Spoofing attacks).

Applications or routines can contain errors, which could allow access to applications running in parallel, or to the operating system. Data sharing causes malicious code to be loaded without the permission of the user. Trojans and viruses come under this category. If these programs are executed in a domain that provides the access rights of the executing user, they may misuse these rights and cause information leakage. Now most of the programs come with checksums and signatures. This feature will reduce such attacks to some extend.

Authorization mechanisms should take care of unauthorized release of the resources in addition to the unauthorized access. The researches in this area are very scanty. A commonly occurring problem is the allocation of previously used space without first erasing it. Any type of swapped memory system is insecure because traces of data are left within that swap file or swap area. If the previously contained information is released to the new owner of the space, it will be a real threat to security **[Anonymous 01]**.

2.1.2. Holes in Operating Systems and Packages

A hole is a feature that allows unauthorized users to gain access or increase their level of access with out authorization. Holes in OSs and packages open a good horizon for the crackers. Any flaw that a cracker can exploit will probably lead to other flaws. Each flaw is a link in the network chain. By weakening one link they can loosen all other links.

Statements or code segments in the available services that cause buffer overflow, unhandled and unexpected combination of inputs, race conditions in a multitasking environment etc. are some examples of common vulnerabilities in operating systems that hackers can make use of **[Its 01]**.

Based on the degree of vulnerability, holes can be classified into 3:

-Holes that cause denial of service (Class C)

-Holes that allow local users with limited privileges to bypass authorization checks (Class B)

-Holes that permit unauthorized access to the network by outside parties (Class A).

Class C holes are almost always operating system based; they often exist in the networking portions of the operating system. Though intruders cannot harm data or gain unauthorized access through this type of holes, they are just nuisances. They often overload the servers and make them inoperable. These attacks are generally called Denial of Service (DoS Attacks).These attacks are common on the Internet. Many of the commonly used DoS attacks are based on high-bandwidth packet floods, or on other repetitive streams of packets.

2.1.3. The Most Common DoS Attacks

A "denial-of-service" attack is characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service. A wide variety of DoS attacks are possible. Examples include blocking of the network traffic, disruption of web services and prohibiting particular persons from accessing certain services **[Loscocco 98]**. Most of these attacks use flooding of the network to achieve the purpose **[Anonymous 01, Cert 99]**.

The most common DoS attacks are Smurf, Fragile and SYN flood **[Cert 96]**. In the "Smurf" attack the attacker broadcasts Internet Control Message Protocol (ICMP) echo requests with return addresses spoofed to the target **[Cert 98]**. This attack has two victims: an "ultimate target" and a "reflector". For each packet sent by the attacker, many hosts on the reflector subnet will respond, flooding the ultimate target and wasting bandwidth for both victims. "Fragile", is a similar attack and uses directed broadcasts in the same way, but uses UDP echo requests instead of ICMP echo requests.

In the SYN flood attack the target machine is flooded with TCP connection requests. The source addresses and source TCP ports of the connection request packets are randomized; the purpose is to force the target host to maintain state information for many connections that will never be completed. SYN flood attacks are usually noticed as the target host (frequently an HTTP or SMTP server) becomes extremely slow, crashes, or hangs. It is also possible for the traffic returned from the target host to cause trouble on routers; because this return traffic goes to the randomized source addresses of the original packets, it lacks the locality properties of "real" IP traffic, and may overflow route caches. This hole exists with in the heart of the UNIX OS or any OS running full fledged TCP/IP over the internet.

Now a days Distributed Denial of Service (DDoS) attacks are also popular. The attacker breaks into hundreds or thousands of machines all over the Internet, installs DDoS software on them and launches coordinated attacks on victim sites from these machines. These attacks typically exhaust bandwidth, router processing capacity, or network stack resources, breaking network connectivity to the victims. Many sites such as Yahoo, Buy, eBay, Amazon, Datek, E*Trade, and CNN have fallen victims to this attack and were unreachable for several hours each **[Zdnet 01, McMillan 07, Sethumadhavan 07]**.

Worms also act as vehicles for DDoS aatacks, as in the case of the fast spreading worms Slammer **[Moore 03]**, Blaster **[Symantec 03-1]** and Sobig **[Symantec 03-2]** which caused wastage of millions of dollars in downtime and IT expenses.

2.2. VULNERABILITY ANALYSIS

Microsoft Windows has a significantly higher share of the web when one counts by the computer. Various surveys reveal that among the current commercial operating systems NT is more popular for attacks **[Netcraft 01]**. According to Attrition, 61% of hacked web sites between August 1999 and May 2001 were running Windows NT/2000, with most of the remainder running variations of UNIX (22% Linux, 7% Solaris, 2% Irix, 8% others) **[Attrition 01]**.

In addition to the above NT suffers from the following security vulnerabilities **[Chalmers 01]**. NT servers lack a single sign on capability for the user's activities. Though Microsoft provides challenge response mechanism for its servers the password encryption is not very strong. For local logins, NT uses an MD4 hash of a 14 character password, with single round (allowing fast password guessing), and does not use a salt (allowing table lookup attacks)**[Visser 97]**.

The effectiveness of memory protection varies from one operating system to another. In most versions of UNIX, it is almost impossible to corrupt the memory of another process. Even though Windows NT provides memory protection, it is susceptible to native VDD-based virus attacks. A program can potentially access memory outside its program space, leading to a system fault which can cause the entire server to fail. Virtual Device Driver viruses have the ability to modify memory, infect or damage files, and directly access both hard drives and floppy diskettes. File level security is provided in Windows NT, only if it's proprietary system NTFS is chosen.

Windows NT Server does allow users to write kernel extensions, and this point of flexibility can also become another point of vulnerability. Windows NT's capability to boot from a floppy diskette as an item of convenience also opens a way for Trojan horse attacks. The way Windows NT schedule the tasks, makes it vulnerable to cpu hog attacks **[Sullivan 01]**

Windows NT Server security is organized by the entire domain, rather than the individual Web server. As a result, if a hacker can manage to control a single server within a domain, he can use that control to access all other servers within the domain **[Mahoney 00]**.

NT provides limited SYN Attack defense, limited firewall capability and has moderate virus susceptibility.

2.3. SECURITY MECHANISMS

Computer security is an enormously difficult problem for which no simple solution exists. An attacker might exploit a design error (holes, weak encryption), a programming error (buffer overflow, unhandled input) a configuration error (accounts with no password, excess rights), or user error (running a Trojan email attachment). Most of the attacks arise from the fact that there is much difference between the user privileges and their actions. Often users get involved in unauthorized retrieval, manipulation and usage of information. The major disadvantage of the existing protection mechanisms is that it is impossible to determine whether such unauthorized information leakage exists. The semantics of the information to be accessed is not considered. Thus it is necessary to find suitable and viable mechanisms according to which the security of a system will not only be decidable, but the system will be secure **[Goscinski 92, Dynamoo 85]**.

2.4. TAXONOMIES OF SECURITY

Military organizations in particular funded the development of provably secure systems. Many taxonomies exist for classifying operating systems and applications based on the level of security offered. These taxonomies are mainly based on access control, resource protection and security violations. The widely accepted taxonomy is the one introduced by Department of Defense U.S (DoD 1983). By this taxonomy most of the commercially available operating systems are marked as less secure with a rating of either C or B. **[Goscinski 92, Radium 02, Bcs 01, Dynamoo 85, Dynamoo 02].** The famous document produced by the US Department of Defense, known as the Orange Book, enshrined various requirements for different levels of trust. By trust, they meant three things: how much you could believe that your system was secure; if it was compromised, how much the system would limit damage; and, after security violation, how well you could trust the logs of the system to tell you what damage had been done. Several trusted operating systems have been built and major UNIX vendors, such as Hewlett-Packard and Sun have trusted (B1-certified) products available **[Radium 01]**. These orderings are generally expensive and hard to administer, and no operating system certified at the B1 level or higher has been successful outside of niche markets. Instead, the broader market has been dominated by systems that are inexpensive, fast, flexible, and feature-rich.

Since modern, widely deployed computer operating systems and applications have well-known fundamental security limitations; users have turned to add-on programs to enhance security. Some of these additions stop attacks before they can succeed. For example, network firewalls **[Cheswick 94]** and the TCP Wrappers package **[Venema**

92] restrict network connections in an attempt to exclude dangerous machines and services. Vulnerability scanners such as SATAN **[Farmer 95]** and Nessus **[Deraison 02]** search and fix known vulnerabilities on a host or network. Packages such as StackGuard **[Cowan 98]** and the Openwall Linux Kernel Patch **[Openwall 01]** prevent many kinds of buffer-overflow attacks from succeeding, either by killing programs that experience stack corruption, or by preventing the foreign stack-resident code from running.

Other additions detect damage after it has occurred. Packages such as Tripwire **[Kim 93]** detect changes to system files by maintaining a set of cryptographically-secure checksums that are periodically recalculated and verified. Virus-protection software, such as Norton Anti Virus **[Symantec 06]** scan local or network storage for signatures of known viruses, allowing users the opportunity to either clean or delete infected files.

The isolation of security functions in a general-purpose operating system is difficult and results in a very large security kernel, which would slow down the performance. A simple solution to ensure security is to monitor the processes that perform access control and logging, giving special emphasis to the security of those functions, and to monitor the processes that manage memory and other resources. Two approaches exist for monitoring and predicting security violations, namely: Misuse detection and Anomaly detection.

The misuse detection system searches for attack signatures and detects already known attacks. But anomaly detection system acts as a supportive subsystem to the operating system for ensuring security by detecting anomalies. An anomaly is the potential possibility of a deliberate unauthorized attempt to access information, manipulate information, or render a system unreliable or unusable. Whenever an object is modified, a privilege used, an authorization granted, or an access to an object or other resources denied, the system sends an audit record to the anomaly detection mechanism. Normally programs make well-defined sequences of system calls, which can be learned by observing the program during its normal behavior. A change from this normal behavior can be predicted as an anomaly.

2.5. INTRUSION DETECTION

An intrusion detection system or IDS is a tool used to detect unauthorized access to a computer system or network. In terms of detection approaches, intrusion detection systems can be broadly divided into two categories - attack-knowledge based and normal-behavior based systems. Attack-knowledge based IDSs apply knowledge accumulated about past attacks to detect future ones. One advantage of attack- knowledge based approach is that it has the potential for very low false positives. Drawbacks, however, include the difficulty of gathering the required information on known attacks and keeping up with new attacks. Normal-behavior based IDSs detect attacks by observing deviations from normal behavior of systems or users. If a deviation is observed, an alarm is raised. A key advantage of normal-behavior based IDSs is that they can detect new, previously unknown attacks. However, a high false alarm rate, due to complex ever-changing normal behavior, is a typical drawback of most normal-behavior based IDSs.

The field of intrusion detection dates back to Anderson's 1980 technical report **[Anderson 80]** and Denning's 1987 intrusion detection model **[Denning 87].** Early work in this field was motivated by the needs of government and the military; the need for better security on the growing Internet, however, has led to numerous research projects and commercial systems. Although there is no set of widely accepted rigorous classifications of intrusion detection systems, they can be broadly classified by what level they monitor (host-based or network-based), and by how they detect attacks (signature, specification, or anomaly). A complete overview of intrusion detection systems is available at Bace **[Bace 99]**. Different taxonomies of the intrusion detection are available at Herve Debar et al. **[Debar 99]**, Axelsson **[Axelsson 00]** and Kaziienko & Dorosz **[Kazienko 04]**.

Since intruders often attack a system through its connections with the outside, modern virus-protection packages also scan web pages and email messages for viruses, and can detect when a program attempts to modify executables (a common method for virus propagation). When acting in this mode, these programs are also acting as intrusion detection systems.

A natural approach to intrusion detection is to monitor network traffic and services. Because of the complexity of network traffic, much attention has been devoted to the problem of what parts of the stream should be examined. One interesting approach is taken by the Network Security Monitor, which focuses on characterizing the normal behavior of a network by examining the source, destination, and service of TCP packets **[Heberlein 90]**. This work served as the basis for Hofmeyr's work on LISYS, a distributed anomaly network intrusion detection system **[Hofmeyr 99]**.

Based on different data sources, IDSs can also be classified into two groups—network-based and host-based. Network-based IDSs, located at choke points (e.g., routers, gateways) in the network to be monitored, capture and analyze network packets for malicious traffic. Host-based IDSs are used to monitor hosts or programs. Another category is application-specific IDS, where IDS is designed for a particular application (e.g., web application, database). As its data source comes from an application which is part of a host, it can be included in host-based IDS category.

In the case of a majority of current intrusion detection systems used to detect human inside attackers, rule-based detection plays an important role. This is due to the fact that the inherent variation of human behavior makes it much harder to extract signatures of inside attacker behaviors compared with those of malicious codes (e.g., viruses). In addition, the other approach—normal behavior-based detection technique, typically has high false alarm rates. Rule-based approaches, in contrast, have the potential to produce low false alarm rates. However, they have difficulty in defining rules which can cover whole attack behaviors because of the diversity of attack behaviors.

There is another category of IDS known as User Behavior based IDSs. This type IDS monitor user behavior. There are two perspectives—general host-based and domain-focused. The data collected by a general host-based IDS represents how users generally behave in a host, while the data collected by a domain-focused IDS is about user behaviors with respect to a particular domain. In the computer world, a domain can be considered as one kind of computer application such as web application, database, or email application.

Owing to the inherent variation of human behavior, the biggest challenge in general host based user-level anomaly IDSs is to decrease false positives without reducing the ability to capture attackers. A number of researchers focus on using different methods to model users via patterns of UNIX commands. DuMouchel **[DuMouchel 99]** uses Bayesian statistics to model command sequences. Schonlau et al. **[Schonlau 01]** investigate and evaluate six statistical approaches for detecting masqueraders. Oka et al.

[Oka 04] propose a novel method called Eigen co-occurrence matrix to model UNIX commands. As an extension of Schonlau's work, Maxion et al. **[Maxion 02-2]** propose a new classification algorithm with improvement on detection rates and false alarm rates.

Like the above researchers, Lane **[Lane 00]** also uses UNIX command sequences as features to model user behavior. Along with computer security issues (storage, speed), he also adopts machine-learning techniques to address these problems in modeling high noise environment, concept drift, skewed class distribution, and variable mis-classification costs. One noteworthy user behavior-level intrusion detection system is NIDES **[Anderson 95]**, which consists of both rule-based and anomaly-based detection approaches. In the anomaly detection part, NIDES creates a set of statistical components to model behavior for individual subjects: users, groups, remote hosts, and the overall system. Parameters of models are dynamically adjusted and specific to each subject. A vector of intrusion detection measures can be used to describe the audited activities. As each audit record arrives, the relevant profiles are retrieved from the knowledge base and compared with the vector of intrusion detection measures. Thus, NIDES evaluates the total usage pattern, not just how the subject behaves with respect to each measure considered singly. Distinguishing features of NIDES are that multiple measures and dynamically adjusted parameters. However, like other user behavior based anomaly IDSs, NIDES has high false positives.

Wisdom & Sense **[Vaccaro 89]** studied historic audit data to produce a forest of rules which describe normal behavior. These rules are then fed to an expert system that evaluates recent audit data and alerts security officers, when the rules indicate anomalous behavior. Majority of the raw information was obtained from computer audit data (e.g., locations, object types, days of the week, person names).

Attack knowledge-based IDSs can be divided into signature-based and rule-based. Signature-based systems analyze data streams for specific patterns (e.g., network packets for substrings correlated with attacks). Rule-based systems compare data to pre-specified rules (e.g., access policy to a specific file).

Signature-based intrusion detection techniques allow for very efficient implementation, and are therefore applied in commercial intrusion detection products such as the ISS RealSecure **[Iss 07]**, WheelGroup Netranger **[Wheelgroup 05]** and Cisco Secure Intrusion Detection System **[Cisco 07]**. The primary advantage of signature-based systems is that they can detect known attacks immediately upon deployment (unlike anomaly-based systems), and they do not need detailed information on the behavior of applications (unlike specification-based systems). The downside is that these systems require frequent signature updates to recognize new attacks, and signatures developed in the laboratory may generate unexpected false positives in the real world.

Most virus and worm scanners also use signature-based techniques. However, conventional signature extraction for novel viruses and worms is an expensive and slow manual procedure that can take hours or even days to complete. Some researchers propose automatic generation of signature of unknown worms without human intervention. EarlyBird **[Sin 03-1, Sin 03-2]**, Autograph **[Kim 04]**, and Honeycomb **[Kreibich 03]** all make use of worm propagation characteristics to automatically generate signatures that can then be used to filter or moderate the spread of worms elsewhere in the network. EarlyBird is based on two key worm behavior characteristics, highly repetitive packet content and increasing traffic volume. Autograph detects worms that propagate by randomly scanning IP addresses. Honeycomb extracts signatures from suspicious traffic caught in honeypots **[Honeypots 07]**.

But with knowledge of a specific domain, it is possible to define rules covering a large majority of attack behaviors in that domain. Therefore, rule-based IDSs are appropriate for those domains which have very clear, critical security requirements such as military, government and finance. In the work of Wisdom & Sense **[Vaccaro 89]**, NIDES **[Anderson 95]**, Haystack **[Smaha 88]** and Emerald **[Porras 97]**, intrusive behavior rules play an important role in detection. The rules defined in Haystack are tied to characteristics and security requirements of the domain (Air Force defense).

Normal behavior-based intrusion detection systems (also called anomaly intrusion detection systems) rely on models of the "normal" behavior of computer systems, users, applications or network usage to detect intruders. Behavior profiles can be built by performing a statistical analysis of historical data or by using rule-based approaches to specify behavior patterns. Anomaly IDSs can be classified into Network based IDS and Host based IDS according to the data sources. They respectively use data transmitted over the network and the data pertaining to the host.

For network anomaly detection, IDSs build profiles based on different parts of packets transmitted over the network. NSM **[Heberlein 90]** uses a four dimensional matrix of which the axes are: source, destination, service, and connection ID. LISYS **[Hofmeyr 99]** is an immunological model of distributed detection which was similar to NSM except that its architecture permits the set of normal network flows to be distributed across a set of hosts. PHAD **[Mahoney 01]** extracts a total of 34 attributes from the packet header fields of Ethernet, IP, TCP, UDP and ICMP. Similar to PHAD, NATE **[Taylor 02]** treats each of the first 48 bytes as a statistical feature, the first 40 bytes of which are in the header part and the latter 8 bytes of which are in the payload part. The work of Krugel et al. **[Krugel 02]** and PAYL by Wang and Stolfo **[Wang 04]** present service-specific intrusion detection systems that combine the service type, length and payload distribution of the request as features. Instead of directly using packet data, some systems like EMERALD **[Porras 97]** reconstruct the network packets and extract features from semantic level data.

Host-based anomaly IDSs use two basic strategies: one is to model program behavior, which is mainly used for detecting viruses and worms; the other is to model user behavior, which is mainly used for insider attacks. In order to model program behavior, a number of researchers have studied system calls **[Ko 94, Somayaji 02]**.

Host-based intrusion detection systems potentially can use many different data sources. Rather than designing custom tools to observe system behavior, most early research in host-based intrusion detection focused on the use of data from audit packages. Audit packages record events such as authorization requests, program invocations, and (some) system calls. This voluminous data is generally written to a special-purpose binary log file. Care is taken to record the data in a secure fashion so that unauthorized users cannot easily conceal their activities. Audit trails are designed to provide forensic evidence for human analysts; however, they can also provide a basis for an automated intrusion detection system. Some of the most sophisticated uses of audit trails were the IDES and NIDES projects, which used SunOS Basic Security Module (BSM) audit data and statistical models to look for unusual patterns of user behavior **[Lunt 92]**. Audit packages by themselves tend to be costly in terms of system performance and storage requirements, and packages such as NIDES only add to the burden. As a result, audit-based intrusion-detection systems have not been widely used outside of government agencies.

Although audit data has been a popular data source, many other sources have been used. Products such as the ISS RealSecure OS Sensor **[Iss 07]** and the free log check package **[Psionic 01]** detect intrusions by scanning standard system logs for attack patterns. Kuperman **[Kuperman 99]** developed a technique for generating application-level audit data through library interposition. Zamboni **[Zamboni 01],** using an internal sensors approach, modified the applications and kernel of an OpenBSD system to report a variety of attempted attacks and other suspicious activity.

One noteworthy product is the CylantSecure intrusion detection system **[Cylant 01]**. Rather than observing system calls, it uses a heavily modified Linux kernel to detect anomalously behaving programs. Published papers **[Elbaum 99, Munson 01]** indicate that Cylant's technology can be used to instrument the source code of arbitrary programs to report as to when different modules are entered and exited. In the CylantSecure product, the behavior of an instrumented Linux kernel is fed into a statistical model that then detects dangerous program behavior and network connections by observing unusual patterns of kernel behavior caused by those programs and connections. Cylant Secure gathers data at the kernel level and performs anomaly detection. It is difficult to make a detailed comparison, though, because little has been published on the algorithms or performance of the system, particularly with respect to false-positives. Audit systems record events after their occurrence. An automated response mechanism based on audit data would generally be triggered after the occurrence of anomalous events, rather than during the events.

Jones and Lin **[JonesK 01]** used sequences of library calls to detect attacks. Many other groups have chosen to use system calls to detect security violations. Ko et al. **[Ko 94]** formally specified allowable system calls and parameters for privileged programs. Wagner and Dean created a two-part prototype system that would dynamically monitor the system calls of a program based on a pre-computed static model derived from the program's source. An anomaly is noted when the program makes a system call that the source would not permit **[Wagner 01]**. This approach is effective in detecting two of the most common forms of attack, namely buffer overflows and trojan code not present in normal binaries; however, it cannot detect attacks that require only existing code.

Endler **[Endler 98]** trained a neural network to detect anomalies in sequences of Solaris BSM audit data. Jones and Li **[Jones 01]** examined temporal signatures constructed from system-call sequences augmented with inter-call timing information. Maxion & Tan **[Maxion 02-1]** compared sequence and markov anomaly detectors using artificial data sets, focusing on their coverage of possible anomalies. Separately, they have also set forth a set of criteria for comparing anomaly detection systems **[Maxion 00]**.

One of the most promising innovations was made by Sekar et al. **[Sekar 01]**. They developed a technique for inferring a finite-state machine using system call data along with the corresponding program counter values. They claimed that their method converges much faster than the sequence method; however, because of the complications introduced by dynamic linking, they ignored the structure of system calls made by library functions. But the sequence method considered this information. So the comparison is somewhat unfair. Nevertheless, the method is promising enough that it deserves a more careful study. **[Somayaji 02]** describes an anomaly detection system pH, which detects changes in program behavior by observing changes in short sequences of system calls. pH can detect buffer overflows, Trojan code and kernel security exploits; however, it has very limited ability to detect masqueraders. Higher-level system behavior has also been studied.

2.6. CONCLUSION

From a study of the literature described in this chapter, it may be concluded that operating systems could be augmented with suitable mechanisms to ensure security. Some useful observations from the research in intrusion detection are the following:

- Unlike misuse detection, which searches for known intrusion patterns, anomaly detection is more popular since it trains normal behavior.
- Normal behavior can be defined in terms of user activities or process execution steps.
- Audit data is slightly expensive in terms of space requirements and speed compared to system call traces to profile process execution steps.

CHAPTER 3

PROCESS PROFILING WITH SYSTEM CALL SEQUENCES

The first step towards detecting the abnormal behavior is to profile the normal program behavior. The method used for profiling should be lightweight so that it will not cause significant loss in performance. It should also not interfere with normal program functioning so that monitored programs can continue to work properly. Most importantly, the mechanism used for profiling normal behavior should be capable of discriminating all possible varieties of security violations.

There are several reasons why system calls are a good basis for a normal program behavior detector. Programs interact with the outside resources through the system call interface. So security violations can be detected by monitoring the system calls generated by the program. The system calls are also relatively easy to observe. Standard methods are available to observe system calls. All system calls invoke the kernel; hence we can observe every process on a system by instructing the kernel to report system call events.

There are different methods to analyze system calls. Many researchers concentrate on how the system call is made. A method for classifying program behavior based on system-call usage could measure system calls in many ways. It could compare the timings of different system calls, or their relative frequencies. It could analyze arguments to specific system calls, or could only look at a subset of all possible system calls. **[Krugel 03]** describes a Bayesian event classification based on system call parameters.

For anomaly detection, many researchers have used the method of analyzing the pattern of system calls generated by a process. In **[Forrest 96]**, short sequences of system calls are used to distinguish normal from abnormal program behavior.

[Somayaji 02] describes two methods to analyze system-call traces: sequences and look-ahead pairs. In both methods, the normal behavior is defined in terms of short sequences of system calls. Both of them use a fixed length window to partition a process's system calls into sequences. In sequence method, a profile of a program's behavior consists of the entire set of sequences produced by that program. The literal contents of each of fixed windows in the profile are recorded, and the set of these sequences are used as the model of normal program behavior. In sequence method, the use of system call sequences of length nine was observed to be optimal.

With the other technique, known as the look ahead pair method, the pairs formed by the current and a past system call are stored in the program's profile. A small fixed size window of system call is used to analyze the system call sequence that slides over each trace, recording which all calls precede the current call within the sliding window. A set of pairs of system calls known as look-ahead pairs are recorded, with each pair representing the current call and a preceding call. For a window of size x, there will be x-1 pairs, one for each system call preceding the current one. The collection of unique pairs over all the traces for a single program constitutes the model of normal behavior for the program.

The look-ahead pair method is very popular for analyzing system call sequences; look-ahead pairs characterize normal program behavior; and the look-ahead pairs method is effective at detecting a variety of anomalies. The sequence method works surprisingly well in comparison to other more sophisticated algorithms. Warrender **[Warrender 99]** compared the sequence method with several others, including a Hidden Markov Model generator and a rule inference algorithm (RIPPER). By analyzing several data sets with each method, she was able to estimate the false and true positive rates of each method. She also roughly measured the execution time required by each method. She observed that the sequence method is as accurate as any other sophisticated algorithm and is much less computationally expensive than other published results **[Hofmeyr 98, Forrest 97]**. However, the look-ahead pair method is both very fast and very easy to implement, and as in **[Forrest 96]**, it is also effective at detecting security violations.

One potential drawback of using fixed length subsequences for detecting intrusions is that the size of the database that contains fixed-length contiguous subsequences increases exponentially with the length of the subsequences. For example, if the number of system calls is 200 and the length of the subsequences is 6, the size of the database is theoretically $200^6 = 64 \text{ X } 10^{12}$. In practice, only normal subsequences are stored, so actual database size is smaller, but still considerably very large.

In 2005 Dae-Ki Kang used 'bag of system calls' representation and compared methods such as supervised learning(SVM) decision trees(C4.5) and Naive Bayesian etc. for detection **[Kang 05]**. He has observed that the bag of system calls can effectively represent intrusions.

Given these constraints, what we needed was a way to compress the traces of system calls into a compact profile that quickly converges to a fixed state, given "normal" program behavior, while still allowing one to detect abnormal behavior as sets of patterns that are not represented in the profile. In addition, this modeling algorithm must be able to capture differences in 200 discrete types of events (the size of the system-call vocabulary); it must also permit fast incremental updates, detect low frequency anomalous events, and have modest memory and CPU requirements.

3.1. THE METHOD

The process profiles are represented using a frequency-based method. The profile consists of the frequencies of system calls in the sequence. Processes behave differently in different conditions. But for each process the number of possible behavior is limited. So the possible sequences for a particular process are finite. With the most straightforward technique, which can be described as the sequence set method, a profile of a program's behavior consists of a number of sequence sets, where each sequence set represent a unique frequency pattern in the entire set of sequences produced by that program.

The sequence set method works surprisingly well in comparison to other more sophisticated algorithms. The experiments showed that the sequence set method was almost as accurate as the best algorithm in any given test, while being much less computationally expensive.

3.2. REPRESENTATION

In our approach, the input sequence is converted into frequency components of the system calls $Xi = \{f_1, f_2, \ldots\ldots, f_n\}$ where n is the total number of possible system calls. The ordering information of adjacent system calls in the input sequence is lost and only the frequency of each system call in the sequence is preserved. Intrusion in this representation is defined according to frequency count of system calls.

More formally, let s_1, s_2, s_3, ..., s_k be the system call trace for a particular process. If i_1, i_2, i_3, ..., i_n be the possible system calls in the sequence. The profile P_{seq} is defined as: $P_{seq} = f_{i1}, f_{i2}, f_{i3}$, ..., f_{in}, where f_{ir} represents the frequency of the system call i_r in the sequence. To make these formal definitions more concrete, it is instructive to work through an example. Figure 3.1 lists the trace of pwd obtained using 'strace pwd'.

```
execve("/bin/pwd", ["pwd"], [/* 35 vars */]) = 0
uname({sys="Linux", node="server.mace.edu", ...}) = 0
brk(0)                          = 0x8ec0000
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=115457, ...}) = 0
old_mmap(NULL, 115457, PROT_READ, MAP_PRIVATE, 3, 0) =0xb7fe3000
close(3)                        = 0
open("/lib/tls/libc.so.6", O_RDONLY)   = 3
read(3,"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 \237v\000"., 512)= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1454835, ...}) = 0
old_mmap(0x755000, 1215644, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x755000
old_mmap(0x878000, 16384, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x123000) = 0x878000
old_mmap(0x87c000, 7324, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x87c000
close(3)                        = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7fe2000
mprotect(0x878000, 4096, PROT_READ)    = 0
mprotect(0x74d000, 4096, PROT_READ)    = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7fe2aa0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0xb7fe3000, 115457)       = 0
brk(0)                          = 0x8ec0000
brk(0x8ee1000)                   = 0x8ee1000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=39542304, ...}) = 0
mmap2(NULL, 2097152, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7de2000
close(3)                        = 0
getcwd("/usr/local/jdk1.1.1", 4096)    = 20
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb7de1000
write(1, "/usr/local/jdk1.1.1\n", 20/usr/local/jdk1.1.1)  = 20
munmap(0xb7de1000, 4096)          = 0
exit_group(0)                   = ?
```

Figure 3.1. A sample system call trace

The parameters passed to system calls and the order in which the system calls appear are not considered. The Trace without parameters is shown in Figure 3.2.

execve, uname, brk, accesss, open, fstat64, old_mmap,close,open,
read, fstat64, old_mmap, old_mmap, old_mmap, close,old_mmap,
mprotect, mprotect, set_thread_area, munmap, brk, open,fstat64,
mmap2, close, getcwd, fsat64, mmap2, write, munmap, exit_group

Figure 3.2. A sample system call sequence

The first step in defining a profile for this trace is to record the frequencies of each system call in the sequence. Figure 3.3 lists frequency of system calls for the 'pwd' example along with the corresponding system call names.

Execve	1
Uname	1
Brk	2
Access	1
Open	3
fstat64	4
old_mmap	5
close	3
read	1
mprotect	2
set_thread_area	1
munmap	2
mmap2	2
getcwd	1
write	1
exit_group	1

Figure 3.3. A sample system call frequency list with system call names

The system calls are encoded as numbers between 0 and 255. The system call names are replaced with the corresponding system call numbers. The corresponding list is shown in the Figure 3.4.

11	1,	122	1,	45	2,	33	1,
5	3,	197	4,	90	5,	6	3,
3	1,	125	2,	243	1,	91	2,
192	2,	183	1,	4	1,	252	1

Figure 3.4. A sample system call frequency list with system call numbers

The list of frequencies is then converted into a sequence with 256 entries where the kth entry in the sequence represents the frequency of system call with system call number k+1. So each sequence will have entries corresponding to system calls 0 to 255.The sample list shown in Figure 3.4 can be represented as a sequence given by Figure 3.5.

[0,0,0,1,1,3,3,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,

0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,0,0,0,0,0,5,2,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,1,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,0, ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,

0,0,2,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,0,0, 0,0,0,0,1,0,0,0,0,0, 0,0,0,1,0,0,0]

Figure 3.5. A sample frequency sequence

These frequency sequences of system calls from different traces of the process are collected and traces with similar frequency components are grouped into sequence sets. Process profiling and anomaly detection using sequence sets are detailed in sections 5.4 through 5.7.

3.3. CONCLUSION

Use of frequency sequence is a compact method for profiling processes using system call traces. Dae-Ki Kang has used frequency-based representation **[Kang 05]** and has observed that the method can effectively represent intrusions. In the 'bag of system calls' representation used by Kang, long sequences of system calls are compressed into short, fixed length sequences called bag containing only frequency values of the system calls. The basic form of representation used in this research is the same as the 'bag of system calls' representation. Incorporation of the new concept called 'sequence sets' in the proposed method simplifies classification of processes, in comparison to Kang's method.

CHAPTER 4

EXPLOITING THE BUFFER OVERFLOWS

Buffer overflows have historically been one of the most popular and best understood methods of exploiting software **[Koziol 94]**. Buffer overflows account for more than 50% of today's vulnerabilities, and this ratio seems to be increasing over time **[Wagner 00]**. Stack overflows have theoretically been around for as early as the C language and exploitation of these vulnerabilities has occurred regularly for well over 25 years. Over the past few years there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail, Linux/FreeBSD mount etc. Even though they are most likely the best-understood and most publicly documented class of vulnerability, stack overflow vulnerabilities remain generally prevalent in software produced today.

There are several different approaches for finding and preventing buffer overflows. These include enforcing secure coding practices, statically analyzing source code, halting exploits via operating system support, and detecting buffer overflows at runtime **[Cowan 03, Zhivich 05]**.

Human code reviews are time-consuming and expensive but can find conceptual problems that are impossible to find automatically. But the probability of missing bugs is high. Code reviews depend on the expertise of the human reviewers, whereas automated techniques can benefit from expert knowledge codified in tools.

Static analysis is performed using an automated tool that scans through the source code with an aim to spot program bugs and suspicious code fragments. This is performed when the program is being compiled. Rather than observing program executions, they analyze source code directly. So not only test cases, but all possible program execution patterns are monitored. From a security viewpoint, this is a significant advantage.

There is a range of static analysis techniques, offering tradeoffs between the required effort and analysis complexity. At the low-effort end are standard compilers, which perform type checking and other simple program analyses. At the other extreme, full program verifiers attempt to prove complex properties about programs. They typically require a complete formal specification and use automated theorem provers. These techniques have been effective but are almost always too expensive and cumbersome to use on even security-critical program.

However, current static analysis tools have unacceptably high false alarm rates and insufficient detection rates **[Zitser 03]**. One disadvantage of using this approach to find errors in source code is that an input revealing the overflow is required, and the input space is generally very large. Therefore, dynamic buffer overflow detection makes the most sense as part of a system that can generate these revealing inputs.

Dynamic buffer overflow detection and prevention is an attractive approach, because fundamentally there can be no false alarms. Tools that provide dynamic buffer overflow detection can be used for a variety of purposes, such as preventing buffer overflows at runtime, testing code for overflows, and finding the root cause of segmentation fault behavior **[Zhivich 05]**.

Several run-time solutions to buffer overflow attacks have been proposed. Operating system patches, such as marking stack memory non-executable, can only protect against a few types of exploits. StackGuard **[Cowan 98]** is a compiler that generates binaries that incorporate code designed to prevent stack smashing attacks. Run-time solutions especially that are based on checking stack memory always incur some performance penalty (StackGuard reports performance overhead of up to 40% **[Cowan 99]**). Other problem with run-time solutions is that while they may be able to detect or prevent a buffer overflow attack, they effectively turn it into a denial-of-service attack. Upon detecting a buffer overflow, there is often no way to recover other than terminating execution.

To overcome the problems of run-time solutions based on stack, David Larochelle **[Larochelle 01]** recommends static code checking so as to detect likely vulnerabilities before deployment. A static analysis tool requires human intervention for accurate detection **[McGraw 04]**. Automating static analysis for buffer overflow is an undecidable problem. He also comments that static checking in general will not provide the complete solution to the buffer overflow problem.

Nguyen et al. **[Nguyen 01]** reported that many privileged programs are inherently capable of doing very limited things, but if the attacker can convince them to fork a general execution shell under their privileged identity, the attacker can gain general privileged access. They observed that a process-oriented model gives better statistical results than user behavioral based model for detecting buffer overflows. They found that process-calling behavior is sufficiently regular for large classes of programs to serve as a good indicator of misbehavior. They also claimed that more than 90% of processes have a deterministic set of authorized children (list of child programs that the program normally forks), and by tracing the sequence of child processes generated, buffer overflows can be detected effectively.

However the detection model described in this dissertation is based on process behavior. This chapter describes the experiments related to buffer overflows that lead to the detection model. Experiments were conducted to generate buffer overflows on simple processes and were extended to other processes. Experiments were also performed to verify whether buffer overflows alter the execution sequence of a process and attempts were made to detect anomalies caused by buffer overflows by analyzing system call sequences.

There are many documents describing buffer overflow exploits and the countermeasures to be taken **[Koziol 04, Balaban 04, Mixter 04, Desai 05, Grover 03, Rider 04]**. The following sections describe how buffer overflow can be used to exploit a program's code so that it will eventually lead to arbitrary code execution. This hack can be used to exploit a program with *suid* set to gain better permissions on a Linux machine.

4.1. MEMORY MANAGEMENT PERSPECTIVE

A buffer overflow is the result of stuffing more data into a buffer than it can handle. Attackers may utilize this often found programming error so as to execute arbitrary code. To understand the buffer overflow and the corresponding attack strategies different components used in the buffer overflow has to be examined in detail. The following sections describe the memory management perspective of buffer overflow.

4.1.1. Buffer

A buffer is defined as a limited, contiguously allocated set of memory. In many languages Buffer overflows are possible because of the lack of inherent bounds-checking mechanisms for buffers. Many languages like C do not have a built-in function to ensure that data being copied into a buffer will not be larger than the buffer size. To overflow is to flow, or fill over the top, brims, or bounds. The exploitation with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows is the main concern of this thesis.

4.1.2. Process Memory

Process memory is divided into three regions: Text, Data, and Stack as shown in Figure 4.1. The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation.

The data region contains initialized and un-initialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the *brk*(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

4.1.3. The Stack Region

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address.

The kernel dynamically adjusts its size at run time. The CPU implements instructions to PUSH onto and POP off of the stack. The boundary of the stack is defined by the extended stack pointer (ESP) register, which points to the top of the stack. In most architectures, SP points to the last address used by the stack. In other implementations, it points to the first free address.

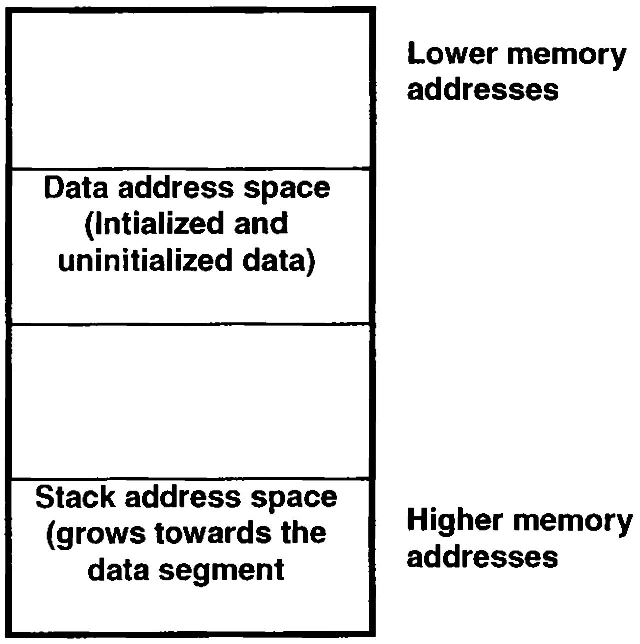
Lower memory addresses
Data address space (Intialized and uninitialized data)
Stack address space (grows towards the data segment
Higher memory addresses

Figure 4.1. Process Memory Regions

The stack consists of logical stack frames that are pushed when a function is invoked and popped when returned. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.

Depending on the implementation, the stack will either grow down (towards lower memory addresses), or up. In the examples described, a stack that grows down is used. This is the way the stack grows on many computers including those using Intel, Motorola, SPARC and MIPS processors. The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack or to the next free available address after the stack. Here the case where SP points to the last address is considered.

In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. It is also referred to as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the tack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Further more, on some machines, such as Intel-based processors; accessing a variable at a known distance from SP requires multiple instructions.

Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, the extended base pointer (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP.

The first thing a procedure must do when invoked is to save the previous FP so that it can be restored at procedure exit. Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog **[Aleph1 98]**. When a function is invoked, EBP the frame pointer is pushed onto the stack. It then copies the current SP onto EBP, making it the new FP pointer. This saved FP can be termed as pointer SFP. It then allocates space for the local variables by subtracting their size from SP.

The memory can only be addressed in multiples of the word size. So, when a buffer whose size is not a proper multiple of word size, is used, the number of bytes equal to the word size or a proper multiple of word size is allocated. So SP should be subtracted considering this word size into account. A view of the stack during the invocation of the function is presented in Figure 4.2.

In Linux and many other operating systems, the stack grows from top to down. When a program calls a function, the return address is pushed and stack pointer is decremented by 4 (32 bits). So definitions of local variables, such as an array, will follow the function return address.

When the stack is descending it can always overwrite the function return address with the contents of the buffer when an overflow occurs. When the function returns, the execution starts from the new overflowed address.

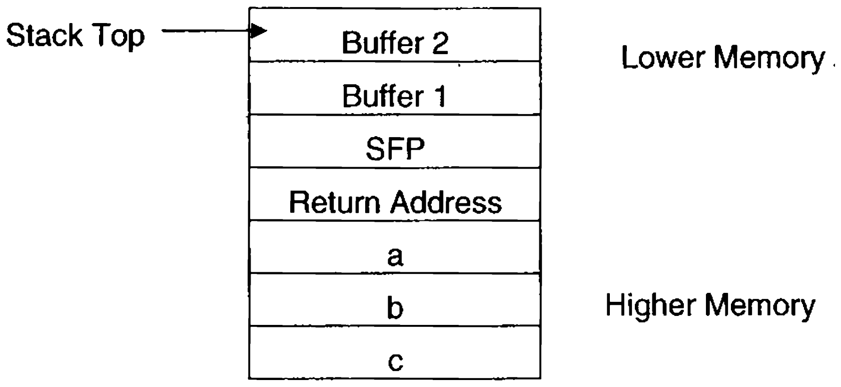

Figure 4.2. An overview of stack during function invocation

4.2. EXPLOITING THE BUFFER OVERFLOW

There are 3 major components in a buffer overflow attack. They are locating overflow potential, hack exploitation and code execution. The most important and complex component is Hack exploitation. The objective of the exploitation part is to divert the execution path of the vulnerable program. We can achieve that via stack-based buffer overflows, heap-based buffer overflows, integer overflows, memory corruption, etc.

Even though we may use one or more of those exploitation techniques to control the execution path of a program, they have to be treated separately. Different vulnerabilities are exploited in different ways. For example, different buffer size and shell code can be used to overflow a particular buffer. For similar buffers, the overflow technique will be same or similar but the shell code used will be different.

Locating the vulnerable code is very easy, especially in a Linux system, since a huge amount of open-source code applications are available for Linux. Some of these applications are in use on almost every Linux system. The strcpy function is the classical example for buffer overflow since it does not check the size of the copied string to ensure that it is within the buffer limits. Many of the programs can be exploited in a similar way with a buffer-overflow hack. Examples of such programs are 'mount' and some early versions of 'innd'. Mount did not check the length of the command-line arguments the user enters. Innd also did not check all of the news message headers, so by sending a specific header, a user could get a remote shell on the server.

Hack exploitation is done in two steps; code insertion and execution redirection. The first one is to find the representation of the code to be executed; this can be done using a simple disassembler. Second step is to make the return address point to the new code inserted. The second part depends on where the program reads the buffer. The buffer can be a mail header or an environment variable or some other parameter whose length goes unchecked. Locating the buffer is not so simple. This is because the address to which the code is loaded, changes often. The address of buffer is determined by guessing. The guessing is continued until the correct address is found. Several ways can be used to make this guessing more efficient. After a few guesses, the right address can be located and the code gets executed.

Once control over the execution path is obtained, the attack code can be executed. A code executer spawns a shell known as shell code. Unlike Exploitation technique, a well designed shell code can easily be reused in other exploits. The inserted code can virtually do everything a computer program can do with the permission of the vulnerable service.

A shell script for buffer overflow is prepared considering the following.

- Fill initial portions of the buffer with garbage
- Place the executable attack code in the buffer
- Calculate the approximate position of the executable in the Stack
- Locate buffer contents which will replace the return address
- Use the approximate address to overwrite return address in the stack

4.3. OVERFLOW EXPLOITATION EXPERIMENTS

Experiments were conducted to create buffer overflows and to execute arbitrary code during the overflow. Following experiments demonstrate the stages in the exploitation of the stack based buffer overflows.

4.3.1. Crossing Buffer Boundary

Consequently, if the person designing the program has not explicitly coded the program to check for oversized input, it is possible for data to fill a buffer, and if that data is large enough, to continue to write past the end of the buffer. In languages, which provide no built-in protection, it is easy to read past the end of a buffer. If the programmer designs code that copies user input into a buffer, it may be possible for a user to intentionally place more input into a buffer than it can hold. This can have a number of different consequences, everything from crashing the program to forcing the program to execute user-supplied instructions. This situation is known as buffer overflow attack.

As a first example to demonstrate buffer overflow a simple function that reads user input into a buffer, and then outputs the user input to stdout is considered. Figure 4.3 lists the example. This program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using strcpy().

Vulnerable C Program

```
int function( char *data)
{
char buffer[100];
strcpy(buffer, data);
return;
}
main ()
{
char overflowstring[160];
gets(overflowstring);
function(overflowstring);
return;
}
```

Figure 4.3. A vulnerable function

The function uses a buffer size of 100. With an overflow string of size 100 or less there will not be any overflow and the program will work correctly. If a string consisting of 120 numbers of character A, is given as input, it will overflow the buffer and start to write over other things stored on the stack. The return address in the stack also will be replaced with 0x41414141 the hexadecimal equivalent of AAAA. When the function exits, attempt will be made to jump to the currently stored return address which is 0x41414141. This address is not a valid address, or is in protected address space, and the program will terminate with a segmentation fault. The core file shows the registers at the time of segmentation fault. During segmentation fault, it was observed that both EIP and EBP contain 0x41414141.

The experiment shows that a buffer overflow allows us to change the return address of a function and to change the flow of execution of the program.

4.3.2. Redirecting Program Flow

It is important to know where to place the code and where the return address is stored **[Ganbold 03]**. If a string of 120 A's are used in the example shown in Figure 4.2, return address will be replaced by 0x41414141. In order to execute an attack, the return address should be replaced by legitimate address. One of the most difficult tasks for executing user-supplied shell code is identifying the starting address of your shell code. Over the years, many different methods have been contrived to solve this problem.

One way to discover the address of the shell code is to guess where the shell code is in memory. In this regard, a pretty educated guess can be made, since for every program, the stack begins with the same address. If the starting address of the stack is known, the distance of the buffer or shell code from the start of the stack can be guessed.

The value of stack pointer can be printed using simple assembly language statements. But exploiting programs in this manner can be tedious. The offset should be guessed correctly and if the guess is wrong, the program crashes. That's not a problem for small programs, but restarting a larger application can take time and effort. More efficient methods are required to find the offsets in large programs.

A technique called the NOP Method can be used to increase the number of potential offsets. No Operations (NOPs) are instructions that delay execution for a period of time. NOPs are chiefly used for timing situations in assembly, or in our case, to create a relatively large section of instructions that do not have any operations. So the beginning of the shell code can be filled with a number of NOPs and this portion is called NOP pad.

When the NOP pad is used before the buffer, the prediction of the starting location of the shell code in the memory need not be 100% correct. Through function return, program control is passed somewhere before the shell code on the stack and through the NOP pad reaches the start of the shell code. In Intel architecture the machine code for NOP is defined as 0x90 **[Eugene 00]**.

4.3.3. Generating the Shell Script

To inject attack code, it should be familiar how an attack is performed and how an attack code is represented. The attack code is some times referred as payload or shell code and is a sequence of machine instructions or op-codes directly understandable by the computer's cpu. The hexadecimal values corresponding to the mnemonic instructions are used **[Maggirrotto 07]**.

Buffer overflows are commonly used to gain root privileges. This is done by attacking a process that is running as root and forcing it to execute (using *execve*) a shell that inherits its permissions. Spawning a root shell is not the only thing that can be performed when a vulnerable program is exploited. This type of local overflow is increasingly popular because local exploit of a program running as non root can be used to get root-level access. There are many exploitation methods other than root shell spawning.

In order to get root privileges, the program should be set to be owned by root, and the suid bit should be turned on. *Setreuid* can be used for this purpose. Now, an ordinary user can use this shell script in the vulnerable program to exploit the buffer overflow and can acquire root access. Figure 4.4 shows the sequence of steps intended to be executed by the vulnerable program during the attack corresponding to *Setreuid* and *Exit.*

```
Set EBX to 0
Set ECX to 0
Set EAX to 0x46
Call INTR 0x80

Set EBX to 0
Set EAX to 0x01
Call INTR 0x80
```

Figure 4.4. Instructions to be inserted during attack

Figure 4.5 lists the Assembly and Hexadecimal codes for *setreuid()*, *exit()*.

```
31,db                      xorl %ebx,%ebx
31, c9                     xorl %ecx, %ecx
b8, 46, 00, 00, 00         mov $0x46,%eax
cd, 80                     int $0x80

31,db                      xorl %ebx,%ebx
b8, 01, 00, 00, 00         mov $0x01,%eax
cd, 80                     int $0x80
```

Figure 4.5. Assembly code corresponding to attack

Most shell code will be converted into a string; the presence of null byte (0x00) in the shell code may be interpreted as a string terminator **[Nomenumbra 05]**. To avoid the null bytes in the shell code, instructions were slightly modified to give a new shell program. The resulting shell code is shown in Figure 4.6.

```
31,db          xorl %ebx,%ebx
31, c9         xorl %ecx, %ecx
31, c0         xor   %eax,%eax
b0, 46         add 46 to eax
cd, 80         int $0x80

31,db          xorl %ebx,%ebx
31, c0         xor   %eax,%eax
b0, 01         add 46 to eax
cd, 80         int $0x80
```

Figure 4.6. Modified assembly code corresponding to attack

The prepared shellcode represented as a string is known as shell script. The shell script is used as a part of the attack script to overflow the buffer. Figure 4.7 shows the prepared shell script. This shell script with slight variations can be used to overflow many of the common programs which use an unbounded buffer as argument.

```
"\x31\xdb\x31\xc9\x31\xc0\xb0\x46\xcd\x80\x31\xdb\x31\xc0\xb0\x01\xcd\x80"
```

Figure 4.7. Shell script

4.3.4. Execution of the Attack

Once the buffer contents are identified the attack can be executed by running the vulnerable program. To perform attack, the buffer of the vulnerable function is loaded with the preplanned attack script. Overview of the attack script is shown in Figure 4.8.

For collecting the normal traces, vulnerable program was executed by passing small strings with size less than 100 as input. A sample trace during the normal execution is given in Figure 4.9.

```
A number of NOPs
Approximate Starting Address of Shellcode
A number of NOPs
Shell Code
  Hex Code for Setreuid(0,0)
  Hex Code for Exit(0)
```

Figure 4.8. Attack script

```
# strace mins
execve("/root/smv/buffprog/mins", ["mins"], [/* 34 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...}) = 0
brk(0)                    = 0x80495ec
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=82863, ...}) = 0
old_mmap(NULL, 82863, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                 = 0
open("/lib/tls/libc.so.6", O_RDONLY)   = 3
read(3,"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3)                  = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 82863)         = 0
fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
read(0, "/x90/x90\n".., 1024)        = 9
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40018000
write(1, "main \n", 6main )         = 6
munmap(0x40018000, 4096)          = 0
exit_group(6)                = ?
```

Figure 4.9. A sample trace during normal execution

To obtain the trace of an abnormal execution, attack was launched with the prepared script. The execution was successfully completed and root privileges were obtained as planned in the attack script. Once root access is obtained, any code can be modified or accessed with root privileges. The corresponding system call trace is shown in Figure 4.10.

```
strace min<in
execve("/root/smv/buffprog/min", ["min"], [/* 34 vars */]) = 0
uname({sys="Linux",node="localhost.localdomain", ...}) = 0
brk(0)                    = 0x8049790
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=82863, ...}) = 0
old_mmap(NULL, 82863, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                  = 0
open("/lib/tls/libc.so.6", O_RDONLY)   = 3
read(3,"\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"., 512)=512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3)                  = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 82863)        = 0
fstat64(0, {st_mode=S_IFREG|0644, st_size=633, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40017000
read(0, "\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90"..., 4096) = 633
setreuid(0, 0)            = 0
_exit(0)                  = ?
```

Figure 4.10. Trace of anomalous execution

The system call trace obtained during normal and abnormal execution is different. The system calls corresponding to the attack script is visible in the abnormal trace. So buffer overflow attacks can be detected by tracing the system calls generated during program execution.

4.4. CONCLUSION

Based on the experiments described above, it was concluded that buffer overflows can be effectively used by attackers to gain control over the system. It was observed that buffer overflows do alter the sequence of system calls generated during execution of processes. Based on the above results, a detection model was developed to detect anomalies, with special emphasis given to buffer overflows.

CHAPTER 5
DETECTION MODEL

Formally, the anomaly detection problem on system call sequences can be defined as follows: Let P = $\{s_1, s_2, s_3, - - -, s_k\}$ *be* a set of system calls where k is the number of system calls in the sequence. Data set D can be defined as a set of labeled sequences {Di , C} where D*i* is an input sequence from the set of all possible P for the particular process and C is a corresponding class label denoting 0 for "normal" label and 1 for "intrusion" label. Given the data set *D,* the goal of the detecting algorithm is to find an efficient classifier.

Many of the anomaly detection systems suffer from large false positives. Two main problems that contribute to the large number of false positives were identified.

The first problem is that, the decision whether an event should be classified as anomalous or as normal is made in a simplistic way. Anomaly detection systems usually contain a collection of models that evaluate different features of an event. These models return an anomaly score or a probability value that reflects the 'normality' of this event according to their current profiles. However, the system is faced with the task of aggregating the different model outputs into a single, final result. The difficulty is the fact that this aggregation is not easy to perform, especially when the individual model outputs differ significantly. In most of the current systems, the problem is solved by calculating the sum of the outputs and comparing it to a static threshold. The disadvantage of this approach is the fact that this threshold has to be small enough to detect malicious events that only manifest themselves in a single anomalous feature (i.e., only one model outputs a high value indicating malicious behavior). This can lead to false positives, because events with many features that deviate slightly from the profile might receive aggregated scores that exceed the threshold.

The second problem of anomaly-based systems is that they cannot distinguish between anomalous behavior caused by unusual but legitimate actions and activity that is the manifestation of an attack. This leads to the situation where any deviation from normal behavior is reported as suspicious, ignoring potential additional information that might suggest otherwise. Such additional information can be external to the system, received from system health monitors (e.g., CPU utilization, memory usage, process status) or other intrusion detection sensors. Consider the example of an IDS that monitors a web server by analyzing the system calls that the server process invokes. A sudden jump in CPU utilization and a continuous increase of the memory allocated by the server process can corroborate the suspicion that a certain system call contains traces of a denial-of-service attack. Additional information can also be directly related to the models, such as the confidence in a model output. Depending on the site-specific structure of input events, certain features might not be suitable to distinguish between legitimate and malicious activity. In such a case, the confidence in the output of the model based on these features should be reduced.

The two problems described above can be mitigated by replacing the simple, threshold-based decision process with a Bayesian network. Instead of calculating the sum of individual model outputs and comparing the result to a threshold, a Bayesian decision process is utilized to classify input events. This process permit to seamlessly incorporate available additional information into the detection decision and to aggregate different model outputs in a more meaningful way.

The frequency list of system calls in system call sequences can be used to represent process behavior and intrusion detection in process executions can be considered as a classification problem of the system call sequences. Intrusion attempts of processes will have more codes usually, than their normal runs. Intrusion runs will have a different frequency distribution than the normal runs. Considering these aspects, attempts were made to explore the application of this representation to anomaly detection tasks using a simple Bayesian classifier.

5.1. BAYESIAN NETWORKS

Traditionally anomaly detection approaches build signatures for known attacks, compare current behavior with those signatures and signal when there is a match. Number of parameters involved, complex relationships among the parameters make simple thresh hold logic less suitable for prediction. So a Bayesian network model is used to predict the probability of an anomaly.

Bayesian Belief Networks (also known as Belief Networks, Causal Probabilistic Networks, Causal Nets, Graphical Probability Networks, Probabilistic Cause-Effect Models, and Probabilistic Influence Diagrams) have attracted much recent attention as a possible solution for the problems of decision support under uncertainty. They are called Bayesian networks because they make use of Baye's rule for probabilistic inference. Although the underlying theory (Bayesian probability) has been around for a long time, the possibility of building and executing realistic models has only been made possible because of recent algorithms and software tools that implement them. Bayesian Network is used to model domains with uncertainty and have proven useful in practical applications such as medical diagnosis and diagnosis of mechanical failures **[Pearl 97, Jensen 96]**.

The Bayesian Network model can represent dependencies among the different objects into its structure. It is made up of a set of variables (nodes) and a set of directed edges between variables **[Jensen 96, Baeza 05]**. Each node has a number of states and a conditional probabilistic table that describes the probabilistic distribution of the states for the corresponding variable given the states of its parent nodes. Graphically a Bayesian network can be described by a directed acyclic graph **[Ye 04]**.

A Bayesian Network can effectively represent the dependence between variables as shown in Figure 5.1 and can give a concise specification of the joint probability distribution.

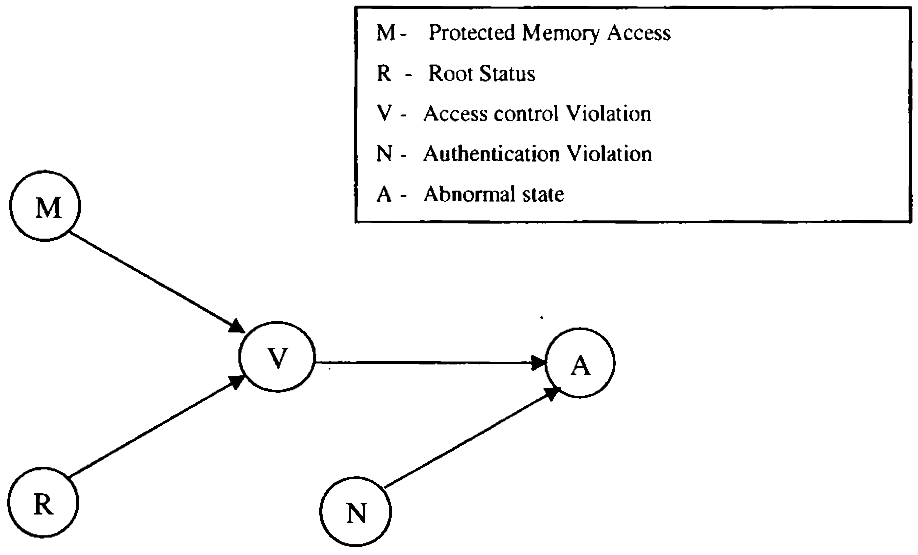
M - Protected Memory Access
R - Root Status
V - Access control Violation
N - Authentication Violation
A - Abnormal state
M
V
A
R
N

Figure 5.1. Representation of dependencies using Bayesian model

Here the nodes represent random variables and the directed links between nodes represent the dependence between the nodes. The Conditional Probability Table (CPT) associated with each node lists the probability that the child node takes on each of the different combinations of values of its parents. If both the child and the parent nodes are binary, they can be denoted with T (true) and F (false) values and a CPT similar to the one listed in Table 5.1 will do.

By the chain rule of probability, the joint probability of all the nodes in the graph given in Figure 5.1 is P(A)=P(N,M,R,V) = (N)*P(M/N)*P(R/M,N)*P(V/R,M,N)

Table 5.1. Conditional Probability Table

R \ M	N	Y
N	8×10^{-4}	1×10^{-6}
Y	3×10^{-2}	2×10^{-4}

By using conditional independence relationships, the equation can be rewritten as

P(A)=P(N,M,R,V)=P(N)* P(M)* P(R) * P(V/ M, R)

The simplifications of second and third terms were possible because of the non dependence of the variables M, R and N. In the last term, simplification was possible because given the parents M and R, V is independent of N. It can be seen that the joint probability can be represented compactly using conditional independence relationships. Here the savings are minimal, but in general, if there are n binary nodes, the full joint would require O(2^n) space to represent, but the factored form would require O(n 2^k) space to represent, where k is the maximum fan-in of a node and fewer parameters makes learning easier [**Murphy 98**].

Any type of anomaly in any application can be predicted if the parameters of importance can be correctly identified. The use of Bayesian Network hides the complexity that arises from the interrelationships among different process parameters that lead to anomalies.

5.2. DETECTION EXPERIMENTS

Initial experiments were conducted in Redhat Linux on simple processes to verify whether buffer overflows can be detected from system call traces. Buffer overflows were created by passing very large strings with intrusion code. System call traces were collected using the command "strace".

Initial experiments were conducted using the simple C program listed in Figure 4.2 as described in section 4.3. System call sequences obtained during normal execution and abnormal execution of the vulnerable program corresponding to Figure 4.9 and Figure 4.10, are shown in Figure 5.2. The portion of the system call trace which is altered in the intrusion trace is shown in bold face.

<u>Normal Execution Trace</u>

Execve, Uname, Brk, Old_mmap, Open,Open,
Fstat64,Old_mmap, Close, Open,Read, Fstat64
Old_mmap,Old_mmap, Old_mmap, Close,
Set_thread_area, Munmap,Mmap2, Read,
Fstat64, Mmap2, Write, Munmap, Exit_group

<u>Intrusion Trace</u>

Execve, Uname,Brk, Old_mmap, Open,Open,
Fstat64,Old_mmap, Close, Open,Read, Fstat64
Old_mmap,Old_mmap, Old_mmap, Close,
Set_thread_area, Munmap,Mmap2,Read,
Setreuid, Exit

Figure 5.2. System Call Trace Analysis

It is observed that the sequences of system calls made by a program during its normal executions are very consistent, and different from the sequences of its abnormal executions as well as the executions of other programs. It is also observed that buffer overflows cause deviation in normal program flow and system call sequence. Therefore a database containing the possible normal sequences with permissible deviations can be used as the definition of the normal behavior of a program and a profile of this database can be used as the basis to detect anomalies. These findings motivated us to search for simple and accurate intrusion detection models based on system call frequencies.

To conduct further experiments, as a commonly used program that is vulnerable to common exploits and buffer overflows, Sendmail daemon was used for studying the normal behavior and to detect anomalous behavior. Sendmail is designed to route email between peers on a network and also to route mails between networks. Many of the functions like crackaddr(), prescan() etc were vulnerable to buffer overflow attacks **[Farrow 03,Cert 03-1,Cert 03-2]**. The syslog intrusions **[Farrow 03]** use the syslog interface to overflow a buffer in sendmail. A very long message overflows a buffer in sendmail, replacing part of sendmail's running image with the attacker's machine code. Though patches are currently available for the most of the vulnerabilities, sendmail and the buffer overflow attacks on sendmail are good cases for experimental study. Sendmail daemon was examined for detection of buffer overflow attacks.

Although the idea of collecting traces of normal behavior sounds simple, there are a number of decisions that must be made regarding how much and what kind of normal behavior is appropriate. Specifically, should we generate an artificial set of test messages that exercises all normal modes of sendmail or should we monitor real user mail with the hope that it covers the full spectrum of normal. This question is especially relevant for sendmail because its behavior is so varied. If we fail to capture all the sources of legal variations, then it will be easier to detect intrusions and be an unfair test because of false positives.

Once the normal profile is defined, the next decision is how to measure new behavior and determine if it is normal or abnormal. The easiest and most natural measure is simply to count the number of mismatches between a new trace and the normal profiles. Ideally, we would like these numbers to be zero for new examples of normal behavior, and to jump significantly when abnormalities occur. In a real system, a threshold value would need to be determined, below which a behavior is said to be normal, and above which it is deemed anomalous.

The sendmail program is sufficiently varied and complex to provide a good initial test, and there are several documented attacks against sendmail that can be used for testing. In order to construct a good classifier, we need to gather a sufficient amount of training data and identify the set of meaningful features. Due to the unavailability of enough varieties of intrusion trace data, further experiments were conducted using data sets available at University of New Mexico **[Unm 02].**

5.3. EXPERIMENTS USING SENDMAIL DATA SETS

For experiments, publicly available system call sequences from UNM data sets were used. The University of New Mexico (UNM) provides a number of system call data sets. Each data set corresponds to a specific attack or exploit. There are data sets corresponding to Linux and Sun systems. Each data set consists of a number of system call traces. Each trace represents a sequence of system call numbers corresponding to the system calls generated. In UNM system call traces, each trace is an output of one program. Sometimes, one trace has multiple processes. In such cases, traces corresponding to different processes are separated and sequence per process is used for analysis. However, most traces have only one process and usually one sequence is created from each trace.

UNM mapping files give the mapping of system call numbers to system call names. There are different mapping files in UNM data sets, corresponding to Sun and Linux.

UNM collection include variety data sets of sendmail and lpr. Synthetic data for sendmail, used in the experiments, were collected at UNM on SUN SPARC stations running unpatched SUNOS 4.1.1 and 4.1.4. The system calls were collected using the strace package, version 3.0.Traces corresponds to the executions of normal sendmail, successful and unsuccessful intrusion attempts on sendmail System calls generated by a process and its children are stored in the same trace. Each trace is a sequence of (process id, system call number). System call numbers are stored in the order in which it is executed. The set include normal traces and abnormal traces. A normal trace consists of several invocations of the sendmail program. The abnormal traces used are from syslog-remote intrusion and syslog-local intrusion. Table 5.2 lists a sample short sequence from the normal data set.

Table 5.2. A short sequence from sendmail normal dataset

3750 5	3752 105	3752 104	3752 104	3752 106

The abnormal traces include local and remote intrusions, each with variety of commands executed during the attack.

5.4. THE APPROACH

The approach is based on two important properties. The first property is that, some unusual short sequences of system calls will be executed when a security hole in a program is exploited. The second property is that variations in the sequence of system calls executed by a program during normal operation are limited. It is assumed that in most cases the first property will be satisfied and if a program enters an unusual error state during an attempted break-in, then the error condition executes a sequence of system calls that is not already covered by the normal database. Also, if an intruder replaces code of running program, it would likely execute a sequence of system calls different from the normal database, and we would expect to see some differences. Finally, it is highly likely that a successful intruder will need to fork a new process in order to take advantage of the system. This fork, when it occurs, should be detectable.

There is a good chance that the latter condition will be met by many programs, simply because the code of most programs is static, and system calls occur at fixed places within the code. Conditions and function calls will change the relative orderings of the invoked system calls but the variations are limited.

However, if an intrusion does not change the system call sequence generated, it will be missed under the current definition of normal. So it is not possible to detect race condition attacks. Typically, these types of intrusions involve stealing a resource (such as a file) created by a program running as root, before the program has had a chance to restrict access to the resource. If the root process does not detect an unusual error, a normal set of system calls will be made, defeating the detection method. This is an important avenue of attack. A second kind of intrusion that is likely to be missed is the case where an intruder uses another user's account. User profiles can potentially provide coverage for this class of intrusions which are not likely to be detectable in root processes. Although the method described here will not provide a fundamentally strong or completely reliable discrimination between normal and abnormal behavior, it could potentially provide a lightweight tool for checking executed code based on frequency of execution.

To achieve reliable discrimination, it should be ensured that the method of flagging sequences as abnormal does not produce too many false negatives or false positives. Most of the exploits studied are found to be very short in terms of the length of time the anomalous behavior takes place. There might be other more appropriate measures than the method that is suggested, especially in an on-line system.

There are two stages to the proposed system. In the first stage, by scanning through the traces of normal behavior a database of characteristic normal patterns (observed sequences of system calls) is built. In the second stage, new traces that might contain abnormal behavior are matched against the normal database. In the current implementation, analysis of traces is performed off-line. In the proposed method, the input sequence is converted into a frequency list of system calls. The system makes use of a new concept known as sequence sets, which is explained in the following paragraphs.

5.4.1. Sequence Sets

System call trace for a particular process is termed as a sequence. A collection of similar sequences is called a sequence set. Sequences that start with the same sequence of system calls will be in the same sequence set if they continue to follow the same sequence of system calls. Certain sequences differ only in the number of times of executing certain subsequences. These sequences, which differ only in the number of execution of subsequence of system calls, will be in the same sequence set. Consider the sequences described in Figure 5.3. The sequences, sequence 1, sequence 2 and sequence 3, differ only in the number of times the subsequence S4, S5, S6 is executed. The system calls and the order in which these system calls are executed are same in all the three sequences. Hence they belong to the same sequence set.

Sequence 1	Sequence 2	Sequence 3
S1	S1	S1
S2	S2	S2
S3	S3	S3
S4	S4	S4
S5	S5	S5
S6	S6	S6
S7	S4	S4
S8	S5	S5
S9	S6	S6
S10	S7	S4
S11	S8	S5
S12	S9	S6
S13	S10	S7
	S11	S8
	S12	S9
	S13	S10
		S11
		S12
		S13

Figure 5.3. A set of system call sequences

Determining which all sequences belong to a particular sequence set is a very complicated process. Determination of sequence sets can be considered as a classification problem. The given input sequences have to be divided into disjoint classes known as sequence sets.

All known algorithms for classification can be applied to the classification of sequences.

5.5. DATA PREPARATION

Each trace in the data set is a collection of several sequences of system calls. Sequences are separated and a frequency chart of system calls for each sequence is prepared. Each sequence is characterized by the start sequence. All sequences with similar starting sequences are grouped into a sequence set. It is assumed that the number of possible normal sequence sets for a particular process is limited. As per the UNM data sets, the number of possible sequence sets for Sendmail is nine and the first seven system calls in the sequence and the frequency of the first system call is used to identify the sequence set to which the particular sequence belongs. Table 5.3 shows a fragment from frequency chart of sequence set1.

5.6. ANOMALY STATUS DETERMINATION

Frequency of individual system calls in the execution trace of a process is used for determining the anomaly status of the particular process. Frequency of each system call in the input execution trace is determined and matched with a normal profile. Details of deviations in frequencies of the input execution trace are fed to the Bayesian network. The Bayesian model computes the anomaly score using system call frequencies as well as prior probability distributions and if the anomaly score is above the threshold value, marks it as an anomalous situation. Pictorial representation of the model used is shown in Figure 5.4.

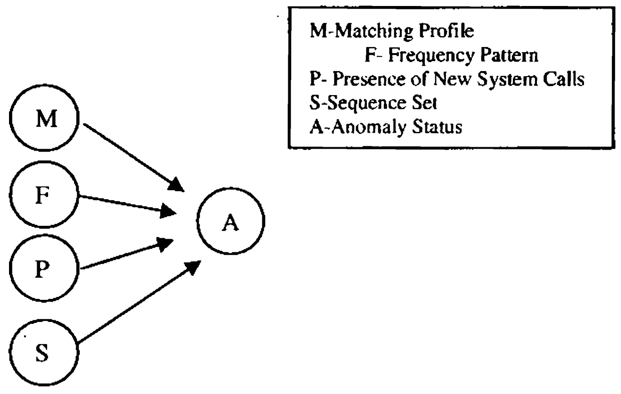

Figure 5.4. Bayesian model

Anomaly status is determined with the help of a Bayesian Network. Frequency chart for the process under consideration is prepared from the input sequence and the corresponding sequence set is determined. A Bayesian Network defines the probability of anomaly for different combinations of system call frequencies. The Bayesian network makes use of a number of model parameters to detect anomalous sequences. The model parameter values are different for different sequence sets.

System call frequencies vary highly in different sequences. Even with in the same sequence set, for certain system calls this variation is unlimited. But for certain system calls, with in the same sequence set, the variation in the frequency is relatively less or limited during normal executions. During a buffer overflow, it is often necessary to insert new code resulting in insertion, deletion or modification of the normal system call sequence. As a consequence, frequency of certain system calls in the sequence deviate from the normal. In most cases, frequencies of system calls can be used to detect anomalous sequence.

System calls are categorized into three groups depending on their frequency variation in anomalous situations. Each model is concerned about a particular category of system calls.

Table 5.3. A fragment from the frequency chart of sequence set1

Process id →						
	System Call Number and Name	3772	3805	3827	3783	3794
Start-sequence		4	4	4	4	4
		2	2	2	2	2
		66	66	66	66	66
		66	66	66	66	66
		4	4	4	4	4
		138	138	138	138	138
		66	66	66	66	66
Frequency Details	1-fork	1	1	1	1	1
	2-read	26	26	26	33	59
	3-write	8	8	8	15	41
	4-open	29	29	29	29	29
	5-close	98	98	98	98	98

5.7. MODEL PARAMETERS

The following section describes the model parameters, their significance and detection mechanism.

5.7.1. Matching Profile

System calls that has limited or no variation with in the same sequence set are considered in the matching profile model. This model approximates and profiles the distribution of frequencies of system calls during normal executions. The goal of this model is to approximate the distribution of the frequencies of system calls of each sequence set and detect instances that significantly deviate from the observed normal behaviour.

For each sequence set there is a normal profile. The profile stores the minimum for normal and maximum possible deviation from the normal for each system call frequency component of the particular sequence set. Each input sequence is matched with the corresponding normal profile. If the frequency components of the input sequence match with the normal profile, with permissible variations, it is treated as a normal sequence by the matching profile model.

5.7.2. Frequency Pattern

Frequencies of certain system calls and subsequences will vary highly even with in the same sequence set. This variation can be considered as normal if this variation is relative to the frequencies of similar system calls. Variation in the frequency of system call Read with system call number 2 in the sequence set1 as shown in Table 5.4 can be considered as example for this case. A fragment from the frequency chart of sequence set4, with identifying sequence "105, 104, 104, 106, 105, 108, 112, 1" is listed to demonstrate the variation in frequencies.

The frequency distribution model captures the concept of a 'normal' system call frequency for such system calls by looking at the relative ranking of the frequency component. It is based on the observation that repeating subsequences will increase the frequency of every system call in the subsequence. The analysis is based only on the relative order of the frequency values and does not rely on the value of the individual system calls.

Table 5.4. A fragment from the frequency chart of sequence set4

System Call Number	1492	1575	1408	1423
2	32	32	12	14
3	15	15	10	11
19	4835	4835	190	670
50	16	16	17	17
78	4814	4814	168	648
104	16052	16052	564	2164
105	9637	9637	344	1304
106	8027	8027	283	1083
108	1610	1610	61	221
112	4830	4830	187	667
128	8	8	10	10

Table 5.5 lists the ranking of the system call frequencies, as used by the frequency pattern model, corresponding to the processes listed in Table 5.4.

5.7.3. Irregularity count /Presence of system calls

Many of the system calls will not appear in the execution sequence of a particular process and will have zero frequency value. This model takes care of system calls absent in all the normal sequences encountered during training phase. The model examines the input sequence for presence of anomalous system calls and outputs an abnormal value if found.

Once the parameters are correctly identified, probability tables can be constructed for predicting the anomaly score. The anomaly score is a value that specifies the extent of the deviation of the received request from the expected profile. It is a compound value that is obtained from the joint probability table. The anomaly score for each request can be in a range from 0.00 to 1.00, where 0.00 represents a completely secure state and 1.00 a sure anomalous state.

Table 5.5. A fragment from the frequency pattern chart of sequence set4

System Call Number	1492	1575	1408	1423
2	8	8	9	9
3	10	10	10	10
19	4	4	4	4
50	9	9	8	8
78	6	6	6	6
104	1	1	1	1
105	2	2	2	2
106	3	3	3	3
108	7	7	7	7
112	5	5	5	5
128	11	11	10	10

5.8. TRAINING

Training involves determination of the sequence sets, system calls used by each of the models, the structure and probabilities associated with each of the nodes in the Bayesian Model. The success of anomaly detection depends on the determination of the right sequence sets and actual probabilities associated with each of the nodes in the Bayesian Network.

System calls used by each of the models and the sequence sets involved are determined by analyzing the variations in system call frequencies and by matching against the identifying sequence.

The Bayesian Network uses a separate node for each model parameter in each sequence set. The joint probability table associated with a node involving variables X_1 to X_k is estimated from the training data as follows

$$P(X_1 = I_1, ---, X_k = I_k) = \frac{N_{X_1 = I_1, ---, X_k = I_k}}{N}$$

where $N_{X_1 = I_1, ---, X_k = I_k}$ is the number of observations in which $X_1, \ldots\ldots, X_k$ are in states $I_1 \cdots\cdot I_k$.

5.9. CONCLUSION

A simple Bayesian model to detect anomalies due to buffer overflows is presented. Frequency sequences from system call traces are classified into sequence sets. The Bayesian model determines whether a particular sequence matches with a sequence set or not. Mismatches are represented as an anomaly score and if the anomaly score is above a threshold value, the particular sequence is termed as anomalous.

CHAPTER 6
PERFORMANCE EVALUATION

A concept prototype was developed and implemented to detect buffer overflows. Sequences were identified with their process-id. Only normal sequences were used for training. Samples were selected using random() function from the list of normal sequences. The prototype was tested using random samples from the list of normal and abnormal sequences.

6.1. PERFORMANCE MEASUREMENT

On analyzing the data set, it is observed that the number of normal profiles is limited. For evaluating the performance, a number of parameters were measured under different conditions. Sequences were identified with their process-ids. Only normal sequences were used for training. Samples were selected using random() function from the list of normal sequences. The prototype was tested using random samples from the list of normal and abnormal sequences.

Performance was measured using cross validation method by varying training and testing data. Only normal datasets were used for training.

Cross validation is a model evaluation method that is better than residuals. The problem with residual evaluations is that they do not give an indication of how well the learner will do when it is asked to make new predictions for data it has not already seen. One way to overcome this problem is to not use the entire data set when training a learner. Some of the data is removed before training begins. Then when training is done, the data that was removed can be used to test the performance of the learned model on new data. This is the basic idea for a whole class of model evaluation methods called cross validation. **[Schneider 97]**

The holdout method is the simplest kind of cross validation. The data set is separated into two sets, called the training set and the testing set. The normal profile is trained using the training set only. Then the data in the testing set is matched against the profile. The testing data is new to detection mechanism. The errors it makes are accumulated as before to give the mean test set error, which is used to evaluate the model. The advantage of this method is that it is usually preferable to the residual method and takes no longer to compute. However, its evaluation can have a high variance. The evaluation may depend heavily on which data points end up in the training set and which end up in the test set, and thus the evaluation may be significantly different depending on how the division is made.

K-fold cross validation is one way to improve over the holdout method. The data set is divided into *k* subsets, and the holdout method is repeated *k* times. Each time, one of the *k* subsets is used as the test set and the other *k-1* subsets are put together to form a training set. Then the average error across all *k* trials is computed. The advantage of this method is that the way in which the data gets divided does not matter much. Every data point gets to be in a test set exactly once, and gets to be in a training set *k-1* times. The variance of the resulting estimate is reduced as *k* is increased. The disadvantage of this method is that the training algorithm has to be rerun from scratch *k* times, which means it takes *k* times as much computation to make an evaluation. A variant of this method is to randomly divide the data into a test set and a training set, *k* different times. The advantage of this metod is that the size of the test set and the number of trials can be independenly selected.

For evaluation 100,000 runs were used. This is because of the high variation observed in the performance of the different runs. Data sequences in each run are divided into training and testing sets. For experiments concerned with training percentage, size of the training data set was selected and all remaining data was used for testing. For all other experiments a total of 10,00,000 random data sequences were selected randomly from all the 10,000 runs. The parameters are totaled and average of all the runs was computed.

The parameters considered for performance measurement were accuracy, detection rate and false positive rate. **[Kang 05]** has defined accuracy, detection rate and false positive rate as follows.

Accuracy is a fraction of accurate identifications.

$$\text{Accuracy} = \frac{\text{True Positive Count} + \text{True Negative Count}}{\text{Total Input Sequences}}$$

Detection rate is a fraction of the intrusions identified.

$$\text{Detection Rate} = \frac{\text{True Positive Count}}{\text{True Positive Count} + \text{False Negative Count}}$$

False positive rate is a fraction of the normal data misidentified as intrusions.

$$\text{False Positive Rate} = \frac{\text{False Positive Count}}{\text{True Positive Count} + \text{False Positive Count}}$$

where true positives are the number of abnormal sequences detected as abnormal, true negatives are the abnormal sequences detected as abnormal, false positives are the normal sequences detected as abnormal and false negatives are the abnormal sequences detected as normal.

6.1.1. Detection of abnormal sequences

The system assumes a threshold value of zero. The system was able to detect all abnormal sequences, keeping the number of false positives small. The false positives are caused by system call sequences which significantly deviate from all examples encountered during the training phase. This is a common problem in intrusion detection practice as pointed out in **[Kang 05]** that the available intrusion data is not quite balanced. In such cases the detection rate and false positive rate will not be optimal. This is due to the huge disparity between the numbers of normal sequences belonging to different sequence sets of the dataset used for evaluation. If this disparity can be removed by selecting all the different varieties of data sequences for training, the number of false positives will be zero.

Most of the false positives were caused by the absence of enough varieties of samples. A typical situation from model2, while considering the frequency ranks of sequence set5 is shown in Table 6.1 and Table 6.2.

Here the false positives were caused due to the specialty of the training.data set pertaining to sequence set5. There are only 2 processes, 1423 and 2905 with a different frequency sequence for the system calls. If both 1423 and 2905 do not appear in the randomly selected training data, the frequency rank profile for sequence set5 will be unique. In this situation frequency sequences corresponding to both 1423 and 2905 will be tested as false positives, since they differ from the unique profile of sequence set5.

Table 6.1. An extract from the frequency chart of set5 showing frequency

System	Frequency Values			
Call Number	PID 1408	PID 1393	PID 1423	PID 2905
2	12	12	14	25
3	10	10	11	16
19	190	73	670	3070
50	17	17	17	17
78	168	51	648	3048
104	564	174	216	1016
105	344	110	130	6104
106	283	88	108	5083
108	61	22	221	1021
112	187	70	667	3067
128	10	10	10	10

Table 6.2. An extract from the frequency chart of set5 showing ranks

System	Frequency Ranking			
Call Number	PID 1408	PID 1393	PID 1423	PID 2905
2	9	9	9	8
3	10	10	10	10
19	4	4	4	4
50	8	8	8	8
78	6	6	6	6
104	1	1	1	1
105	2	2	2	2
106	3	3	3	3
108	7	7	7	7
112	5	5	5	5
128	10	10	10	11

A similar situation occurs in sequence set1 in model1. The increase in the number of false positives was caused by the lack of sufficient number of variety data in the training data set. Table 6.3 and Table 6.4 show the situation. This is because of similarity of the processes appearing in sequence set1. There are only two processes 3783 and 3794 in sequence set1 with a different frequency profile. All other processes belonging to sequence set1 has a fixed pattern of frequency values for all system calls. In sequence set1 there are only two system calls, which are handled by the frequency pattern model or model2. So if both 3783 and 3794 do not appear in the training data, profile for sequence set1 will contain fixed frequency values for all system calls with non zero frequency values and are handled by model1 itself. In this special situation, even system call 2 and 3 which were supposed to be handled by model2 will be handled by model1. Model1 signals both 3783 and 3794 as false positives, due to the difference in the frequency values of system call 2 and system call 3.

Table 6.3. An extract from the frequency chart of set1 showing frequency

System Call Name	System Call Number	Frequency Values			
		PID 4176	PID 4187	PID 3783	PID 3794
Read	2	26	26	33	59
Write	3	8	8	15	41

Table 6.4. An extract from the frequency chart of set1 showing ranks

System Call Name	System Call Number	Frequency Ranking			
		PID 4176	PID 4187	PID 3783	PID 3794
Read	2	1	1	1	1
Write	3	2	2	2	2

6.1.2. Effect of Training Ratio on performance

In the first set of experiments, performance was measured by varying training percentage. The prototype was tested using random samples from the list of normal and abnormal sequences. Figure 6.1 shows the corresponding chart.

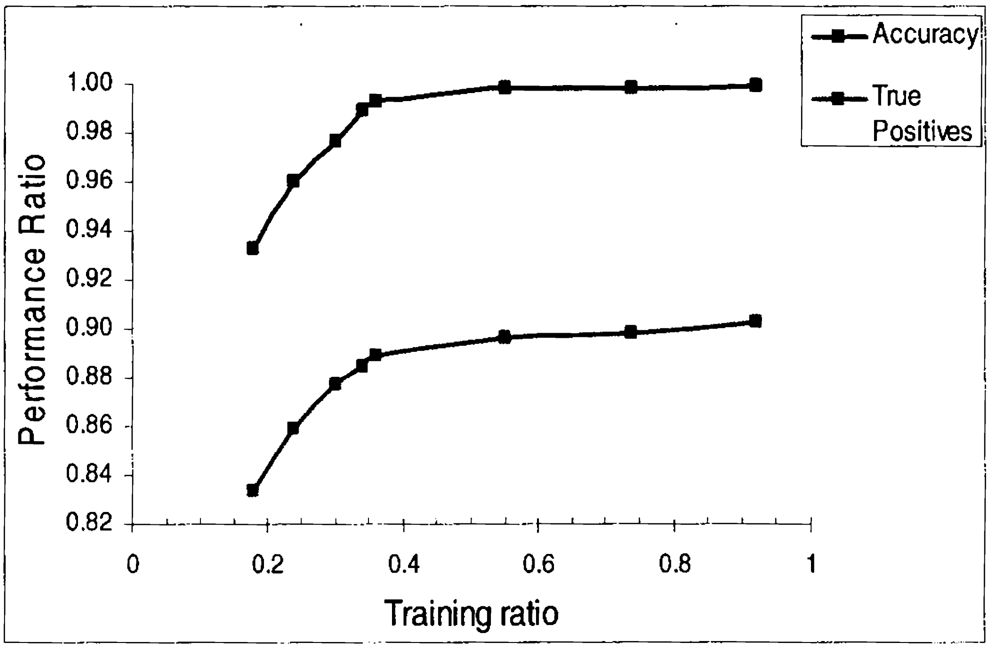

Figure 6.1. Performance evaluation 1

6.1.3. Comparison of False Positives

The number of false positives is a very important criteria to determine the success of the method. Following sections describe the comparison of false positives under different conditions.

6.1.3.a. Effect of Training Ratio

The effect of training ratio on false positives was studied. Figure 6.2 shows the corresponding chart.

6.1.3.b. Effect of True Positive Ratio

Next set of experiments compared the effect of true positive ratio on false positives. In order to vary the true positive ratio, more abnormal samples were added to the testing data by replicating the process sequences.

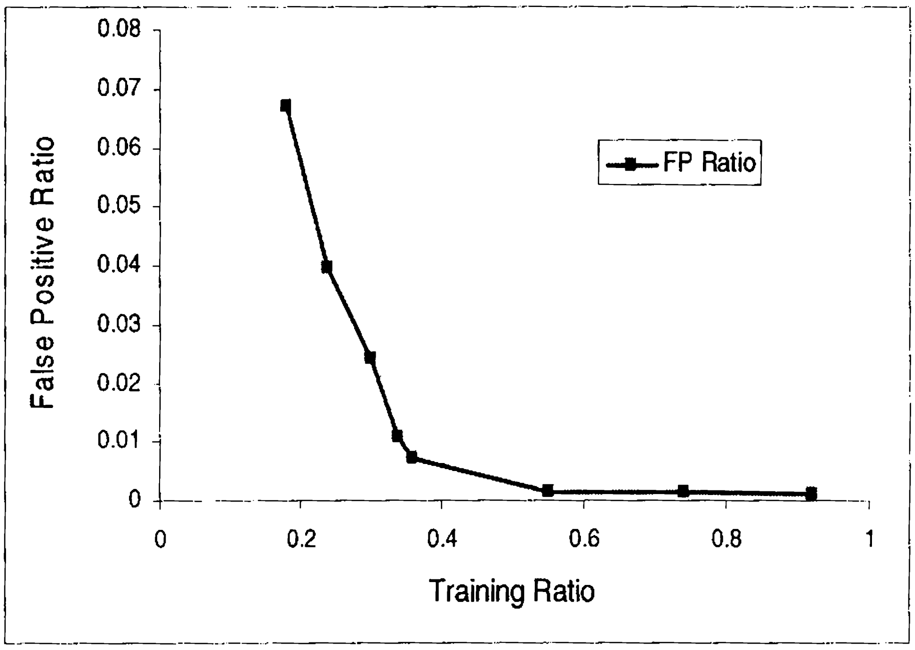

Figure 6.2. Performance evaluation 2

Figure 6.3 shows the effect of true positives on false positives when training percentage is varied. The first set used 50% and the second set used 80% of random data samples for training.

Figure 6.4 shows the effect of true positives on false positives when random samples are used for testing.

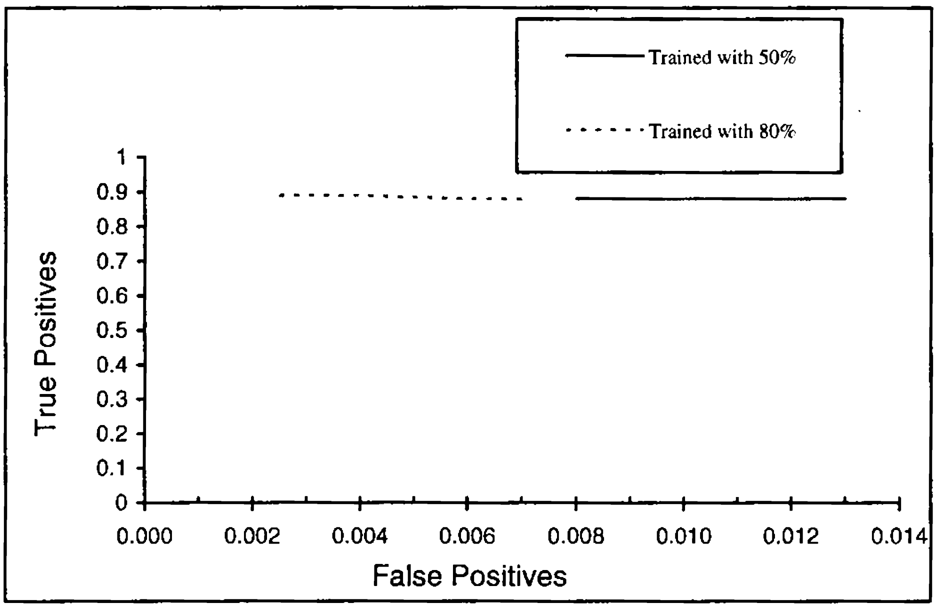

Figure 6.3. Performance evaluation 3

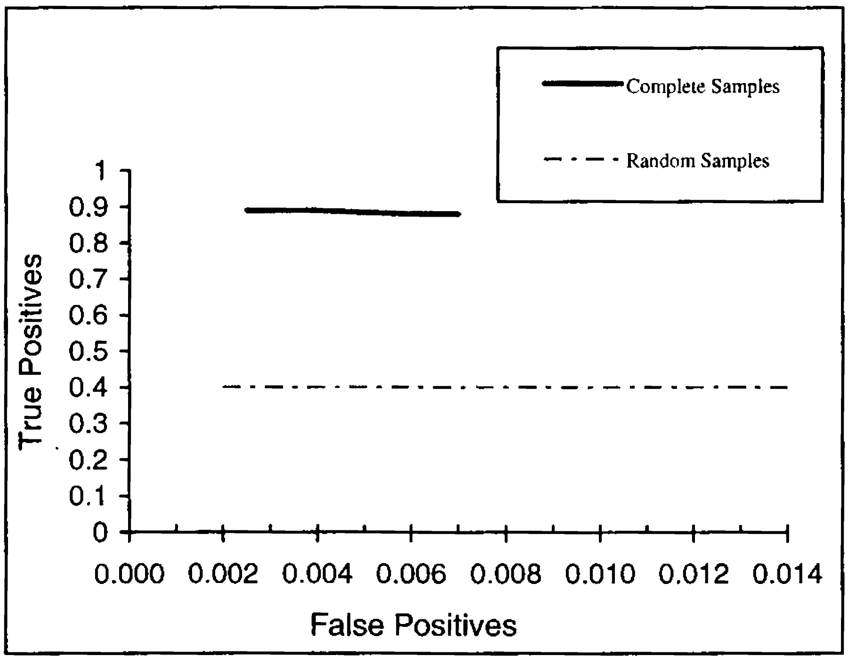

Figure 6.4. Performance evaluation 4

In the first set complete data samples were used for testing. Second set used randomly selected data for testing. Training percentage was 80% for both set of experiments.

The emphasis has been on determining the success of the approach and efficiency issues were not much considered.

6.2. PERFORMANCE COMPARISON

[Kang 05] compares the different methods to analyze frequencies of system calls for anomaly detection. A comparison of the sequence-set method with other methods is listed in Table 6.5.

From Table 6.5 it is clear that the suggested method is highly efficient in terms of accuracy, detection rate and false positive rate, to detect anomalies in UNM Synthetic sendmail data set. Due to the unavailability of other buffer overflow data sets, it could not be tested for other datasets.

Table 6.5. Comparison based on performance percentage

UNM Synthetic Sendmail	Naive Bayes	C4.5	RIPPER	SVM	Logistic Regre-ssion	Sequence Set
Accuracy	20.21	94.87	94.33	95.68	95.41	99.93
Detection Rate	92.00	40.00	48.00	40.00	64.00	100
False Positive Rate	84.97	1.15	2.31	0.28	2.31	0.16

Figure 6.5 represents the performance parameters graphically.

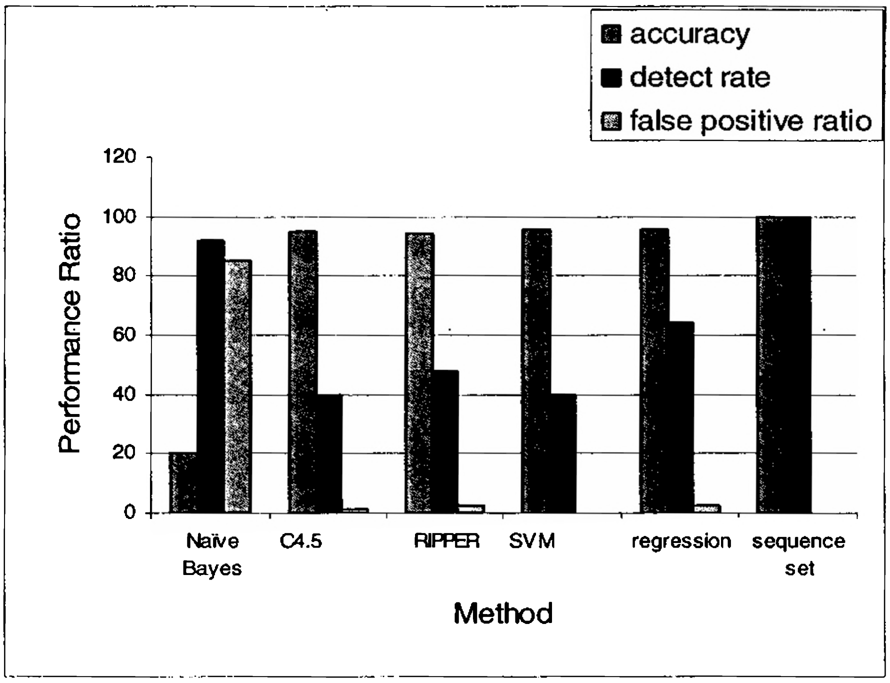

Figure 6.5. Performance comparison 1

[Zhang 05] presents a comparison of HMM method and subsequence analysis with two sequence lengths. The comparison of the methods is given in Figure 6.6.

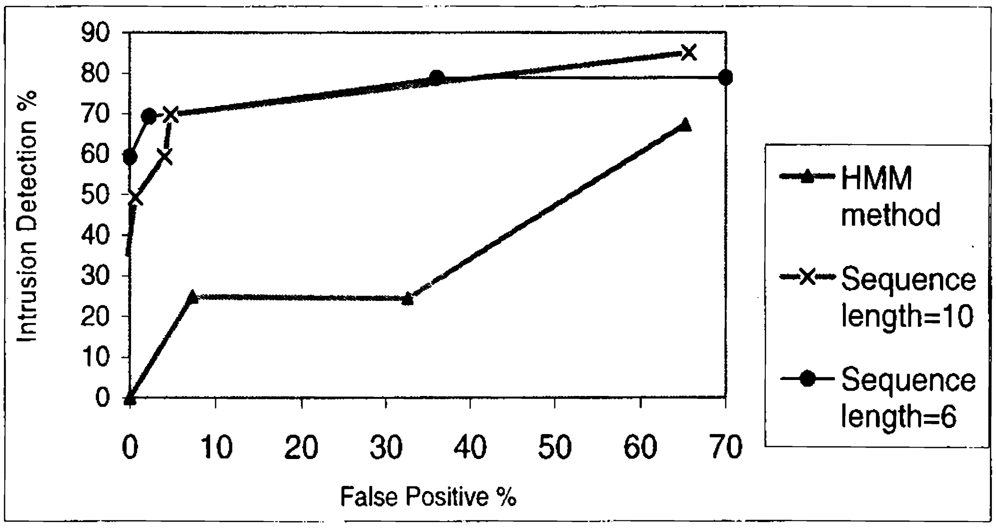

Figure 6.6. Performance comparison 2

False positive rate for sequence set method is very low in comparison to the other methods (Figure 6.2 to 6.4). So it is not included in the graph. The highest detection rate achieved for any false positive ratio for the different methods from Figure 6.6 was compared with detection rate for sequence set based method (proposed method). The results are shown in Table 6.6.

Table 6.6. Comparison of Highest detection ratio

Subsequence method with Sequence length=6	Subsequence method with Sequence length=10	HMM method	Sequence Set based method
62 %	50 %	5 %	100 %

So the sequence set method is superior to the other methods such as HMM method and subsequence methods in terms of high detection ratio and low false positives.

6.3. CONCLUSION

The sequence set method was evaluated using cross validation method using UNM data sets and found to have good performance parameters such as 100% detection rate, high accuracy and low false positive rate. Available evaluation details of HMM method and subsequence methods were used to compare the proposed method's performance. The sequence set method was found to be superior in terms of all the performance parameters used.

From the previous paragraphs it is clear that use of sequence sets and their analysis using simple Bayesian network is highly effective and efficient.

CHAPTER 7

DISCUSSION

Although the detection mechanism is both remarkably efficient and effective, it cannot be perfect in all respects.

The Detection Mechanism (DM) is useful in providing powerful security data for review, with its capability to determine which applications are running malicious software. It is expected that this database of applications will be extremely helpful in performing an audit of firewall rules and monitoring for policy compliance. By setting the anomaly detector, it is possible to quickly watch all the applications and locate vulnerabilities and security violations. Vulnerability assessment and anomaly detection will help the administrators to alert on or block, threats. A defense center can aggregate information from all the sensors and this data can be used to secure the network centrally. Using the data gathered from the anomaly detector, the defense center can raise or lower the "relevance" of an event based upon the target's importance and its susceptibility to the attack. In addition to a variety of alerting methods, administrators can block traffic via inline Intrusion Sensors, or via third-party firewalls, switches, and routers. They can also remedy vulnerabilities via third-party patch and configuration management solutions.

DM is implemented for offline processing of system call traces. It analyses system call traces and determines the anomalies during process executions. The current prototype is implemented in C in windows XP. It is not easily portable from windows.

Currently the training phase is separate and does not incorporate new sequences into normal profile.

Detailed discussions follow, along with comparison with other approaches.

7.1. SELECTION OF TRAINING SAMPLE

Before an anomaly detection system can be effective it must have a model of normal behavior. Normal profile is collected during training. The training data should be very carefully selected so as to contain all different categories of sequences. The next new code path may generate a dozen new sequences or none at all.

Perhaps the simplest way to ensure that Detection Mechanism (DM) has profiles of normal program behavior would be for software developers to distribute default profiles of normal program behavior. These synthetic normal profiles could be easily generated by running some or all of a program's regression test suite. If DM detects anomalous program behavior relative to such a profile when a program is being used properly, then the program's test suite is not comprehensive enough. Over a period, DM will replace many of these profiles with ones that are specialized to the usage patterns of a host. These profiles would generally be smaller than the default synthetic normal profiles and would restrict program behavior to those code paths that are actually used on a given machine.

7.2. EFFECT OF THRESHOLD

For example, a per-profile anomaly threshold could be tightened as a profile stabilized. In sequence-based methods, a change in user behavior has much effect on the correct threshold. But in this method, the effect of threshold is very steady. Zero threshold is a good choice since it considers all which deviates from normal as anomalous even if the variation is very small. To get an accurate performance, a threshold of zero can be chosen.

7.3. ADAPTING TO NEW SEQUENCES

Detection Mechanism could also be improved through the addition of a user space daemon to manage process profiles and regulate status responses. Such a daemon could note when new programs are run and use site-specific policies to determine whether it should be allowed or not. It could periodically scan the profiles directory for normal or almost-normal profiles that are likely to generate false positives.

The Mechanism never forgets program behavior even if a given behavior was encountered only once. To mitigate this limitation, the daemon could prune profiles to remove entries that hadn't been recently used. By correlating anomalies with network connections or log entries, a monitoring daemon could also decide whether a few scattered anomalies indicate that the system is under attack. It could then use this information to trigger a customized response.

7.4. PORTABILITY

DM is implemented in windows. With a few changes, the mechanism should also be able to run in Linux kernel. Because most UNIX variants use monolithic kernels and support similar system call interfaces, it should be straightforward to port core analysis routines to such systems.

DM captures the essence of a program's interactions with the outside world by observing its system calls. Hence it cannot be adopted on systems that do not support a traditional system call interface.

7.5. OFFLINE PROCESSING

When frequency component of system calls are used for analysis, the monitored process should terminate, in order to collect the final sequence of samples generated. So only the offline processing of system call traces is adopted.

With slight modifications, the method can be applied to on-line processing. The frequency components at each vulnerable point can be gathered and the same may be used for determination of anomalies.

7.6. LIMITATIONS

Similar to other monitoring approaches using system calls, the capability of the suggested approach is limited by the information contained in the system calls. The parameters passed to the system calls are also not considered. System calls contain information only about the limited events that occur. Only events that use system calls will appear in the system call sequence of a particular process. This restricts the capabilities of the monitoring system.

7.7. IMPROVEMENTS

Instead of capturing the behavior of entire applications, a better approach may be to monitor the execution of application components. Stillerman et al. **[Stillerman 99]** have shown that sequences of CORBA method invocations can be used to detect security violations; in a similar manner, it should be possible to detect security violations in Windows applications.

The violations that can be detected by the anomaly detector depend on the set of normal traces. The set of trace policies restrict the behavior of programs beyond that enforced by the protection mechanisms built in the operating system. Further research is needed to identify a methodology for determining whether a given set of normal traces is adequate for a system.

Similar to other intrusion detection approaches, the new approach is not a panacea to the intrusion problem. Attacks that do not produce state changes (e.g., passive wiretapping) or that require massive behavioral analysis can not be detected. This approach also assumes integrity of the system call data. Thus, attacks that involves spoofing, which produce the same system call trace but from a different source, may not be detected.

Last, the new approach is a detection approach, which raises an alarm when an intrusion occurs. This approach can, at best, detect security violations, and it is up to the system administrator or security officer to deal with each detected violation.

7.8. COMPARISON

The proposed solution is a simple, accurate and systematic approach compared to current approaches for monitoring the security of a system.

7.8.1. Misuse Detection

In misuse detection, the goal is to identify actions (or misuse signatures) that represent intrusive activities and to check for occurrences of these actions in the audit trails. Expert system rules, state-transition diagrams, and patterns in Petri networks describe misuse signatures.

The specification-based approach can be thought of as the dual of misuse detection. A misuse signature describes undesired behavior in a system while a trace policy describes the desirable behavior of a subject. In particular, new approach focuses on the desirable behavior of security-critical programs (e.g., privileged programs) in a system. One way to specify the desirable behavior of a program is to enumerate the operations the program needs to perform in order to accomplish its function.

A misuse detector matches a signature with the whole system trace to identify intrusions while an analyzer in a specification-based execution monitor parses the trace of a subject to determine whether the subject conforms to a trace policy. Although matching of different signatures can be distributed over multiple hosts, each misuse detector requires the whole system trace. In a distributed system with many hosts, the whole system trace would be huge and cannot be processed by a misuse detector in real time. In the suggested approach, an analyzer monitors the execution of a particular subject; only the audit records associated with the subject are needed by the analyzer.

In misuse detection, signatures are mostly driven by previous attacks or known vulnerabilities. Although possible, it is not intuitive to encode a policy as misuse signatures. The new approach is more policy-oriented; a trace policy for a subject is specified based on the functionality of the subject and the system security policy. Therefore, it can succeed in catching attacks that exploit unknown vulnerabilities in programs.

7.8.2. System Call Based Methods

When compared with the widely used fixed-length contiguous subsequence models, the system call representation explored in this dissertation is somewhat simple. It may be argued that much more sophisticated models are available that take into account the identity of the user or perhaps the order in which the calls were made. But the experiments show that a much simpler approach may be adequate in many scenarios. The results of experiments described in the previous sections show that it is possible to achieve near perfect detection rates and false positive rates using a data representation that discards the relationship between system call and originating process as well as the sequence structure of the calls within the traces.

It is possible to achieve accurate anomaly detection using fixed-length contiguous subsequence representation of input data. In this approach, the detector will find anomalous subsequences right after they are executed depending on user-specified thresholds. A weakness of the subsequence based approach **[Somayaji 98]** is that small changes in user behavior can result in very different patterns of system calls. In general the rate of novel sequences goes down; yet for all programs, there are discontinuities when usage patterns change. A profile that has "almost settled down" is not "almost stable"; the appearance of even a few novel sequences means that previously unseen code paths are being executed.

The proposed model has advantages that learning is faster, memory requirements are significantly lower and simple Bayesian model discriminates normal sequences and abnormal sequences.

7.9. CONCLUSION

This thesis describes a new approach to security monitoring. The main idea is to detect buffer overrun vulnerabilities by specifying the desirable behavior of security-critical programs in a system and to monitor their executions for behavior inconsistent with the specifications.

The approach aims at building process profiles with system call frequencies and to detect anomalies by measuring deviations from the process profile. Aspects of program

T492

behavior that are security-relevant were identified and a number of model parameters for describing the desirable behavior of programs were developed. The detection model uses novel ideas from probability, and provides a way to enhance accuracy of detection. The method is demonstrated by applying it to a simple vulnerable process and sendmail.

The model is very suitable for detecting well-known attacks and trivially modified attacks under time and space constraints. If the anomaly detector needs to be built in real-time and the built system must be as light as possible to be able to work over limited resources, the new approach will be a perfect fit because the generated detector is simple and powerful to detect well known attacks.

Since no effective mechanism has been determined for dynamic detection and prevention of buffer overflows, obtaining information about the occurrence of buffer overflow itself is very important. The suggested method performs offline analysis of system call traces to detect presence of anomalies due to buffer overflows. Taking into account the security relevant information such as the time of attack, the function which is used for attack, the user who executed the vulnerable program and the attack code used, the system administrator can take precautionary measures to prevent further attacks, once the anomaly is signaled. The anomaly detector presented, is a security mechanism that moves us a step closer to a self- defending system.

UNIVERSITY LIBRARY • COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY • COCHIN 682 022

CHAPTER 8
FUTURE WORK

This chapter discusses suggestions for future research in this area. The design of a detection mechanism is presented and implementation of an offline prototype for a single host is discussed. The implementation serves as a proof of concept of the new approach.

The new approach can be deceived by mimicry attacks **[Wagner 02]**, if the attacker knows the intrusion detection mechanism. Future work in this area should be focused on addressing this problem.

The implementation was tested using the data sets available at University of New Mexico. Due to the difficulty in collecting sufficient varieties of intrusion data pertaining to buffer overflow, the model could not be tested in a real environment for complex cases. More testing in a real environment is needed. One future effort would be to implement an on-line distributed monitoring system based on the suggested design. The system should trace the system calls generated, update the frequency components in real time and perform real time analysis. The distributed monitoring should work on a large distributed system and have efficient communication mechanisms. In particular, it is important to evaluate the performance of the monitoring system in a very large distributed system. The test should clarify how large a system the monitoring system can handle. The performance evaluation should include CPU time, memory requirements, network bandwidth consumed by the system, and the response time of normal users. Criteria for distributing the load of the analyzers among different machines are also needed.

REFERENCES

1. [Aleph1 98] Aleph1, Smashing The Stack For Fun And Profit , May 1998, http://www.phrack.org/issues.html?issue=49&id=14

2. [Anderson 80] James P. Anderson. Computer Security Threat Monitoring And Surveillance. Technical Report, James P. Anderson Co:, Fort Washington, PA, 1980.

3. [Anderson 95] D. Anderson, T. Frivold, And A. Valdes. Next-Generation Intrusion Detection Expert System (NIDES): A Summary. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, May 1995. www.mcafee.com/au/local_content/white_papers/wp_intruvertnextgenerationids.pdf

4. [Anonymous 01]Anonymous, Maximum Security, Sams Net, Techmedia, 2001.

5. [Attrition 01] "OS Statistics: August 99 To Present ",accessed in June 2001. http://www.attrition.org/mirror/attrition/os.html#ALL

6. [Axelsson 00] Stefan Axelsson. Intrusion Detection Systems: A Taxomomy And Survey. Technical Report 99-15, Dept. Of Computer Engineering, Chalmers University Of Technology, March 2000.

7. [Bace 99] Rebecca Gurley Bace, Intrusion Detection, Macmillan Technical Publishing, 1999.

8. [Baeza 05] Ricardo Baeza, Berthier Ribeiro, Modern Information Retrieval , Pearson Education, 2005.

9. [Balaban 04] Murat Balaban, Buffer Overflows Demystified, , accessed in June 2004. http://www.enderunix.org/docs/eng/bof-eng.txt

10. [Bcs 01] A Risky Business - Protecting Online Insurers, http://www.bcs.org.uk/review/2001/html/p182.htm

11. [Cert 03-1] CERT® Advisory CA-2003-07 Remote Buffer Overflow in Sendmail, dated June 09, 2003.http://www.cert.org/advisories/CA-2003-07.html

12. [Cert 03-2]CERT® Advisory CA-2003-25 Buffer Overflow in Sendmail, Dated September 29, 2003. http://www.cert.org/advisories/CA-2003-25.html

13. [Cert 96] CERT® Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks". http://www.cert.org/advisories/CA-1996-21.html

14. [Cert 98] CERT® Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, http://www.cert.org/advisories/CA-98.01.smurf.html

15. [Cert 99] CERT® Advisory CA-1999-17 Denial-Of-Service Tools , http://www.cert.org/advisories/CA-1999-17.html

16. [Chalmers 01]. A comparison of the security of Windows NT and Unix, www.ce.chalmers.se/staff/jonsson/nt-vs-unix.pdf

17. [Cheswick 94] William R. Cheswick and Steven M. Bellovin. Firewalls and Internet Security. Addison-Wesley Pub Co., 1994

18. [Cisco 07] Cisco Systems, Cisco Secure Intrusion Detection System, accessed in January 2007. http://www.cisco.com/univercd/cc/td/doc/product/iaabu/csids/index.htm

19. [Cowan 98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, Proceedings of the 7th USENIX Security Symposium San Antonio, Texas, January 26-29, 1998.

20. [Cowan 99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. Protecting Systems from StackSmashing Attacks with StackGuard. Linux Expo.May 1999. (Updated statistics at http://immunix.org/StackGuard/performance.html)

21. [Cylant 01] Software Systems International,Cylant Division Home Page.Http://www.Cylant.Com, January 2001.

22. [Debar 99] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. Computer Networks, Volume 31, Issue 8,pp: 805-822, April 23, 1999.

23. [Deitel 00] Harvey M. Deitel, Paul J. Deitel, "An Introduction to Operating Systems", Prentice Hall of India, 2000.

24. [Denning 87] Dorothy E. Denning, An Intrusion-Detection Model, IEEE Transactions on Software Engineering, SE-13 (2), pp : 222-232, February 1987.

25. [Deraison 02] Renaud Deraison et. al.,The Nessus Project. http://www.nessus.org

26. [Desai 05] Suhas Desai ,Buffer overflow attacks, 8/9/2005, http://www.e-articles.info/e/a/title/Buffer-Overflow/

27. [DuMouchel 99] W. DuMouchel, Computer Intrusion Detection Based on Bayes Factors for Comparing Command Transition Probabilities, Technical report, National Institute of Statistical Sciences (NISS), USA, 1999.

28. [Dynamoo 02] Orange Book Summary, accessed in March 2002. http://www.dynamoo.com/orange/summary.htm

29. [Dynamoo 85] " Department Of Defense Standard ,Department Of Defense Trusted Computer System Valuation Criteria" ,December 1985. http://www.dynamoo.com/orange/fulltext.html

30. [Elbaum 99]Sebastian Elbaum and John C. Muson. Intrusion Detection through Dynamic Software Measurement. in Proceedings of the 1st Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, The USENIX Association, April 9-12, 1999.

31. [Endler 98] D. Endler. Intrusion Detection: Applying Machine Learning to Solaris Audit Data. In Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98), pp: 268-279, Scottsdale, AZ, December 1998.

32. [Eugene 00] Eugene@subterrain.net , Architecture Spanning Shellcode, 9/5/2000, http://www.groar.org/expl/immediate/spanning.html.

33. [Farmer 95] Dan Farmer and Wietse Venema. Satan Home page,1995. http://www.porcupine.org/satan/

34. [Farrow 03] Rik Farrow, Security Musings, Login –The Magazine of USENIX AND SAGE , Vol 28, number 3, June 2003.

35. [Forrest 96] S.Forrest.A Sense of Self for UNIX Processes.In Proceedings of the IEEE Symposium on Security and Privacy, pp 120 –128,Oakland, CA, May 1996.

36. [Forrest 97] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. Computer Immunology, Communications of the ACM, Volume 40, Issue10, pp: 88-96, October 1997.

37. [Ganbold 03] Ganbold, Problems with Sample Buffer Overflow Exploit Solved, 02 October 2003. http://archieve.cert.uni_stuttgart.de/vuln_dev/2003/10/msg00006.html

38. [Goscinski 92] Andrzej. Goscinski, Distributed Operating Systems, The Logical Design - Addison Wesley, 1992.

39. [Goyal 03] Bharat Goyal, Sriranjani Sitaraman and S. Venkatesan, A Unified Approach to Detect Binding Based Race Condition Attacks, In Proceedings of the International Workshop on Cryptology and Network Security (CANS) in

conjunction with the International Conference of Distributed Multimedia Systems (DMS)) Florida, USA, pp: 605-610 September 2003. also at www.utdallas.edu/~venky/publications/race.pdf.

40. [Grover 03] Sandep Grover, Buffer Overflow Attacks and their Counter Measures, Linux Journal, 10-3-2003, http://www.linuxjournal.cpm/6701.

41. [Heberlein 90] L.T. Heberlein, G.V. Dias, K.N. Levitt, and B. Mukherjee. A network security monitor. In Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy, pages 296-304, 1990.

42. [Hofmeyr 98] S. A. Hofmeyr, S. Forrest, A. Somayaji,Intrusion Detection Using Sequences of System Calls, Journal of Computer Security, Volume 6, pp:151–180, 1998.

43. [Hofmeyr 99] Steven A. Hofmeyr, An Immunological Model of Distributed Detection and its Application to Computer Security, PhD thesis, University of New Mexico, 1999.

44. [Honeypots 07] Honeypots, accessed in January 2000. http://www.honeypots.net

45. [Iss 07] Internet Security Systems Inc. RealSecure Internet, accessed in January 2007. http://www.iss.net/products/product_sections/Intrusion_Detection_.html.

46. [Its 01] Secure Operating System, INFOSEC 99, ANS T1.523-2001, Revisions of the Telecom Glossary 2000, accessed in March 2001. http://www.its.bldrdoc.gov/projects/t1glossary2000/_secure_operating_system.html.

47. [Jensen 96] Finn V. Jensen, An Introduction to Bayesian Networks, Springer, 1996.

48. [Jones 01] Anita Jones and Song Li. Temporal signatures for intrusion detection. In the Proceedings of the 17th Annual Computer Security Applications Conference, pp252-264, New Orleans, Louisiana, December 10-14, 2001.

49. [JonesK 01] Anita Jones K. and Yu Lin. Application Intrusion Detection using Language Library Calls. In Proceedings of the 17th Annual Computer Security Applications Conference, pp 442-449, New Orleans, Louisiana, December 10-14, 2001.

50. [Kang 05] Dae-Ki Kang, Doug Fuller, Vasant Honavar, Learning Classifiers for Misuse and Anomaly Detection Using a Bag of System Calls Representation,

ISI 2005, LNCS 3495, pp: 511–516, 2005.
http://www.cs.iastate.edu/~honavar/Papers/isi05.pdf

51. [Kazienko 04] Przemyslaw Kazienko, Piotr Dorosz, Intrusion Detection Systems (IDS) Part 2 - Classification; Methods; Techniques, :July 23, 2004. http://www.windowsecurity.com/s/IDS-Part2-Classification-methods-techniques.html

52. [Kim 04] H.A. Kim and B. Karp. Autograph: Toward Automated Distributed Worm Signature Detection. In USENIX Security Symposium, pages 271–286, 2004. http://www.cs.cmu.edu/~bkarp/autograph-usenixsec2004.pdf

53. [Kim 93] Gene H. Kim, Eugene H. Spafford, The design and implementation of tripwire: A file system integrity checker. Technical Report CSD–TR–93–071, Purdue University, November 1993.

54. [Ko 94] Calvin Ko, George Fink, Karl Levitt., Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring , In Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, FL, pp: 134-144, December 5-9, 1994.

55. [Koziol 94] Jack Koziol, David Litchfield,David Aitel,Chris Anley,Sinan noir Eren,Neel Mehta,Riley Hassell, Stack Overflows, "The Shellcoder's Handbook: Discovering and Exploiting Security Holes", Wiley, 1994 .

56. [Kreibich 03] C. Kreibich and J. Crowcroft, Honeycomb - Creating Intrusion Detection Signatures Using Honeypots, In the Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II), November 2003. www.cl.cam.ac.uk/~cpk25/publications/honeycomb-hotnetsII.pdf

57. [Krugel 02] C. Krugel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In SAC '02: Proceedings of the 2002 ACM Symposium on Applied Computing, 2002.

58. [Krugel 03] Christopher Krugel, Darren Mutz, William Robertson, and Fredrik Valeur: Bayesian Event Classification for Intrusion Detection, pp:14-23, ACSAC 2003, Las Vegas, NV, USA.

59. [Kuperman 99] [Benjamin A. Kuperman and Eugene Spafford. Generation of application level audit data via library interposition. Technical Report CERIAS TR 99-11, COAST Laboratory, Purdue University, West Lafayette, IN, October 1999.

60. [Lane 00] T. Lane, Machine Learning Techniques for the Computer Security Domain of Anomaly Detection. PhD thesis, Purdue University, 2000. www.cs.unm.edu/~terran/downloads/pubs/thesis.ps.gz

61. [Larochelle 01] David Larochelle , Statically Detecting Likely Buffer Overflow Vulnerabilities. http://lclint.cs.virginia.edu/usenix01.pdf

62. [Litchfield 03] David Litchfield, Defeating the stack based Buffer Overflow Prevention Mechanism of Microsoft windows 2003 server, Proceedings of Black Hat, 8-9-2003. http://www.blackhat.com/presentation/bh_asia_03/bh_asia_03_litchfield.pdf

63. [Loscocco 98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, John F. Farrell, The Inevitability of Failure:The Flawed Assumption of Security in Modern Computing Environments , Published in the Proceedings of the 21st National Information Systems Security Conference, pages 303-314, October 1998. also at http://12.110.110.204/selinux/papers/inevitability.pdf

64. [Lunt 92] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, H.S. Javitz, A. Valdes, and T.D. Garvey. A real-time intrusion detection expert system (IDES) | final technical report. Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

65. [Maggirrotto 07] Denis Maggirrotto David Litchfield Explanation of a Remote Buffer Overflow Vulnerability, 22-04-2007. http://milw0rm.com/papers/160

66. [Mahoney 00] Matt Mahoney, "Computer Security: A Survey of Attacks and Defenses", September 11 2000. http://cs.fit.edu/~mmahoney/ids.html

67. [Mahoney 01] M. Mahoney, P. Chan. PHAD, Packet Header Anomaly Detection for Identifying Hostile Network Traffic, Technical Report CS-2001-04, Florida Institute of Technology, 2001.

68. [Maxion 00] Roy A. Maxion and Kymie M. C. Tan. Benchmarking Anomaly-Based Detection Systems, pp 623-30, In the Proceedings of the International Conference on Dependable Systems and Networks, IEEE Computer Society Press, New York, NY, June 25-28, 2000.

69. [Maxion 02-1] Roy A. Maxion, Kymie M. C.Tan. Anomaly Detection in Embedded Systems, IEEE Transactions on Computers,Volume 51, Issue 2, pp: 108-120, February 2002.

70. [Maxion 02-2] R. Maxion,T. Townsend, Masquerade Detection using Truncated Command Lines, In DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, Washington, DC, 23-26 June 2002. www.cs.cmu.edu/afs/cs.cmu.edu/user//www/pubs/MaxionTownsend02.pdf

71. [McGraw 04] Gary McGraw, Static Analysis For Security, pp: 32-35, IEEE Security and Privacy 2004. http://www.cigital.com/papers/download/bsi5-static.pdf

72. [McMillan 07] Robert McMillan, Second Google Desktop Attack Reported Vulnerability to a Little-Known Web-based Attack could Allow an Attacker to have Access to any Data Indexed by Google Desktop, Infoworld, February 23, 2007. http://www.infoworld.com/article/07/02/23/HNsecondgoogledesktopattack_1.html

73. [Milankovic 00] Milankovic, Operating System Concepts and Design, McGrawHill, 2000.

74. [Mixter 04] Mixter, Writing Buffer Overflow Exploits – A Tutorial For Beginners, accessed in June 2004. http://mixter.void.ru/exploit.html

75. [Moore 03] David Moore, Inside the Slammer Worm,pp:33-39, IEEE Security & Privacy, 2003. also at http://www.cs.ucsd.edu/~savage/papers/IEEESP03.pdf

76. [Munson 01] John C. Munson and Scott Wimer. Watcher: The missing piece of the security puzzle. In Proceedings of the 17th Annual Computer Security Applications Conference, New Orleans, Louisiana, December 10-14, 2001.

77. [Murphy 98] Kevin Murphy, A Brief Introduction to Graphical Models and Bayesian Networks, 1998.

78. [Netcraft 01]. "Web server survey results", accessed in June 2001. http://www.netcraft.com/survey/

79. [Nomenumbra 05] Nomenumbra, Practical windows and Linux Shellcode Design,http://www.milw0rm.com/papers/11

80. [Oka 04] M. Oka, Y. Oyama, H. Abe, and K. Kato. Anomaly Detection Using Layered Networks Based on Eigen Co-occurrence Matrix. In RAID 2004 Proceedings, volume 3224 of LNCS, pages 223–237. Springer-Verlag, 2004. http://www.itsec.gov.cn/webportal/download/2004-Anomaly%20Detection%20Using%20Layered%20Networks%20Based%20on%20Eigen%20Co-occurrence%20Matrix.pdf

81. [Openwall 01] Solar Designer. Linux kernel patch from the openwall project,2007. http://www.openwall.com/linux

82. [Osborn 06] Russ Osborn, Moderen Buffer Overfow prevention Techniques, How they work and Why they don't, 13/4/2006 http://www.cs.hmc.edu/~nike/pubic_htm/courses/security/s06/projects/russ.pdf.

83. [Pearl 97] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, 1997.

84. [Porras 97] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabled Responses to Anomalous Live Distrubances. In National Information Systems Security Conference, 1997.

85. [Psionic 01] Psionic Software. Logcheck version 1.1.1. http://www.psionic.com/abacus/logcheck, January 2001.

86. [Radium 01] National Computer Security Center. Trusted Product Evaluation Program (TPEP) Evaluated Products By Rating, January 2001. http://www.radium.ncsc.mil/tpep/epl/epl-by-class.html

87. [Radium 02] Department of Defense Standard -Department of Defense Trusted Computer System Evaluation Criteria , DoD 5200.28-STD,December 1985, accessed in March 2002. http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html

88. [Rider 04] Ghost Rider, Introduction to Buffer Overflows, accessed in 2004. http://mixter.void.ru/exploit.html

89. [Schneider 97] Jeff Schneider, Cross Validation, 7 February 1997. http://www.cs.cmu.edu/~schneide/tut5/node42.html.

90. [Schonlau 01] M. Schonlau,W, DuMouchel,W. Ju, A. Karr, M, Theus,Y. Vardi, Computer Intrusion: Detecting Masquerades, Statistical Science, Volume 16, Issue 1,pp:1–17, 2001. also at http://www.niss.org/technicalreports/tr95.pdf.

91. [Sekar 01] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati, A Fast Automatonbased Method for Detecting Anomalous Program Behaviors, In IEEE Symposium on Security and Privacy, 2001. http://documents.iss.net/whitepapers/Proventia.pdf

92. [Sethumadhavan 07] Rajesh Sethumadhavan, Yahoo Messenger 8.1 Address Book Buffer Overflow ,17 July 2007,http://www.net-security.org/vuln.php?id=4278

93. [Singh 03-1] S. Singh, C. Estan, G. Varghese, and S. Savage, Automated Worm Fingerprint ing. In SOSP03:19th ACM Symposium on Operating Systems Principles, August 2003, http://www.cs.ucsd.edu/~savage/papers/OSDI04.pdf

94. [Singh 03-2] S. Singh, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Real- time Detection of Unknown Worms. Technical Report CS2003-0761, University of California, August 2003. also at www.cs.unc.edu/~jeffay/courses/nidsS05/signatures/savage-earlybird03.pdf

95. [Smaha 88] S. Smaha, Haystack, An Intrusion Detection System, In the 4th Aerospace Computer Security Applications Conference, pp :37 – 44,12-16 Dec 1988. also at www.acm.org/crossroads/xrds2-4/intrus.html

96. [Somayaji 02] Anil Somayaji. Operating System Stability and Security through Process Homeostasis. PhD thesis, University of New Mexico, 2002.

97. [Stillerman 99] Matthew Stillerman, Carla Marceau, and Maureen Stillman, Intrusion Detection for Distributed Applications, Communications of the ACM, Volume 42, Issue 7, pp:62-69,July 1999.

98. [Sullivan 01] . "Eamonn Sullivan","NT vulnerable to attack on CPU" , accessed in December 2001. http://www.zdnet.com/eweek/news/1216/18ent.html#top

99. [Symantec 03-1] W32.Blaster.Worm, Symantec, 2003. http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html

100. [Symantec 03-2] W32.Sobig.F@mm, 2003. http://www.symantec.com/security_response/writeup.jsp?docid=2003-081909-2118-99

101. [Symantec 06] Symantec. Norton Antivirus 2006, 2006. www.symantec.com/nav/nav_9xnt/

102. [Taylor 02] C. Taylor, J. Alves-Foss, An Empirical Analysis of NATE - Network Analysis of Anomalous Traffic Events, In NSPW '02: Proceedings of the 2002 New security paradigms Workshop, 2002.

103. [Unm 02] Computer Immune Systems - Data Sets and Software, accessed in June 2004. http://www.cs.unm.edu/~immsec/systemcalls.htm

104. [Vaccaro 89] H. Vaccaro, G. Liepins, Detection of Anomalous Computer Session Activity, In Proceedings of IEEE Security and Privacy, May 1989.

105. [Venema 92] Wietse Venema. TCP WRAPPER: network monitoring, access control, and booby traps. In Proceedings of the 3rd UNIX Security Symposium, 1992.

106. [Visser 97] Jos Visser, "On NT security", 5 May 1997, accessed in March 2001. http://www.osp.nl/infobase/ntpass.html

107. [Wagner 01] David Wagner and Drew Dean. Intrusion detection via static analysis. In Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.,pp 175–187, Oakland, California, USA, May 2001.

108. [Wagner 02] David Wagner, Paolo Soto,Mimicry Attacks on Host-Based Intrusion Detection Systems, Proceedings of the 9th ACM Conference on Computer Communications Security, pp: 255-264,Washington, DC,USA,2002.

109. [Wang 04] K. Wang and S. Stolfo. Anomalous Payload-based Network Intrusion Detection. In Proceedings of RAID'04, September 2004. http://www1.cs.columbia.edu/ids/publications/raid4.pdf

110. [Warrender 99] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In Proceedings of the 1999 IEEE Symposium on Security and Privacy, IEEE Computer Society, pp 133-145, Los Alamitos, CA, 1999.

111. [Wheelgroup 07] WheelGroup Corporation. Brochure of the Netranger Intrusion Detection System, accessed in June 2005. http://www.wheelgroup.com

112. [Ye 04] Nong Ye , Mingming Xu, Probabilistic Networks with Undirected Links for Anomaly Detection, accessed in June 2004. www.itoc.usma.edu/workshop/2000/Abstracts/WA1_2.pdf

113. [Zamboni 01] Diego Zamboni, Using Internal Sensors for Computer Intrusion Detection. PhD thesis, Purdue University, August 2001.

114. [Zdnet 01] " Web attacks: FBI Launches Probe", http://www.zdnet.com/zdnn/stories/news/0,4586,2435149,00.html?chkpt=zdhpnews01http://www.infoworld.com/article/07/02/23/HNsecondgoogledesktopattack_1.html

115. [Zhang 05] Xiaoqiang Zhang, Zhongliang Zhu, Pingzhi Fan ,Intrusion Detection Based on Cross-Correlation of System Call Sequences, Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05), pp: 277-283, Hongkong, November 2005.

116. [Zhivich 05] Michael Zhivich,Tim Leek, Richard Lippmann,Dynamic Buffer Overflow Detection, *2005 Workshop on the Evaluation of Software Defect Detection Tools* 2005, Chicago, IL, June 2005.

117. [Zitser 04] Misha Zitser,Richard Lippmann,Tim Leek, Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT FSE 2004: 97-106. 2002.

PUBLISHED WORK OF THE AUTHOR

1. An Anomaly Detection System in Operating Systems for Ensuring Security, Proceedings of the 2002 International Conference on Security Management, SAM`02, Las Vegas, June 2002, pp 411-415.

2. Bayesian Networks for Detection of Anomalous Situation, Proceedings of the International Conference of Resource Utilization & Intelligent Systems, INCRUIS `06, Kongu Engineering College, Erode, India, January. 4-6, 2006, pp 739-743.

3. Frequency Analysis of System Calls for Buffer Overflow Detection, Proceedings of International Conference on Network Security, ICONS `07, Sengunthar Engineering College, Erode, India, January.29-31, 2007, pp 563-570.

4. Buffer overflows: Exploitation and Detection, Proceedings of International Multiconference of Engineers and Computer Scientists, IMCES 2007, Hong Kong, March 21-23, 2007, pp 918-923.

5. Process profiling using Frequencies of System Calls, Proceedings of Second International Conference on Availability, Reliability & Security, ARES 2007, Vienna April 10-13, 2007, pp 473-479.

6. Anomaly Detection Using System Call Sequence Sets, Journal of Software (JSW), Volume:2, Issue:6, December 2007, ISSN:1796-217X, pp 14-21.

UNIVERSITY LIBRARY
COCHIN 682 022
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY