A Lightweight Real-time Host-based Intrusion Detection System

by

Kevin E. McDonald

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

Author ....
Department of Electrical Engineering and Computer Science
May 25, 2001

Certified by ....
Richard P. Lippmann
Senior Scientist, MIT Lincoln Laboratory
Thesis Supervisor

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

A Lightweight Real-time Host-based Intrusion Detection System

by

Kevin E. McDonald

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

During recent years, the number of computer attacks has increased dramatically. These attacks can have costly effects including downtime for servers, the theft of valuable corporate data, and web site vandalism. Intrusion Detection Systems (IDS's) have become an important tool for detecting and preventing these computer attacks. IDS's may be either network-based, or host-based. Host-based IDS's provide monitoring capabilities for the system they are running on, while network-based systems analyze network traffic to detect attacks in the traffic to many machines. Host-based systems offer the advantage of being able to directly monitor the state of their host, while network-based systems need to infer the state of monitored machines.

This thesis describes a host-based Intrusion Detection System for UNIX systems. The IDS, called ReaLLite, is capable of detecting attacks in real-time and is lightweight enough to run in the background of any system. ReaLLite runs as a user-process rather than with root privileges, thereby eliminating most potential security risks involved with its use and simplifying installation. ReaLLite monitors process data, network packet statistics and connection information, and file system inode data using standard UNIX commands and utilities. With this information, ReaLLite is capable of detecting buffer overflow attacks, Denial of Service attacks, and port scan probes.

ReaLLite was tested using 7 attacks from the 1999 DARPA Intrusion Detection Evaluation and with 5 newer attacks. ReaLLite successfully detected 11 of these 12 attacks. All of the 8 buffer overflow attacks were identified, as was the port scan probe, and two of the three Denial of Service attacks. The only missed DoS attack disabled the monitored host before ReaLLite could respond. ReaLLite was also successfully ported to the Linux Operating System, and detected the two Linux attacks it was tested with.

Thesis Supervisor: Richard P. Lippmann
Title: Senior Scientist, MIT Lincoln Laboratory

Acknowledgments

I would like to first and foremost thank my thesis supervisor, Richard Lippmann, for his support of my research and for keeping me headed in the right direction. I would like to thank Seth Webster, Josh Haines, Dave Fried, Lee Rossey, and Kavita Baball for reviewing drafts of this thesis and providing valuable feedback. I would also like to thank Rich, Seth, Dave, Josh, Lee, Jesse Rabek, Raj Basu, and the rest of the intrusion detection group for their helpful comments, ideas, and suggestions. Finally, I would like to thank my parents and family for all of their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The explosive growth of the Internet has created an equally explosive increase in attacks on Internet hosts. The term "attack" refers to a wide range of activities. These include disrupting legitimate users from accessing a computer or service and gaining illegal access to privileges on a computer system. Current security practices to counter attacks involve the use of firewalls, encryption, and authentication. These countermeasures are often inadequate, however. Recently, numerous high profile web sites such as Microsoft, eBay, and Yahoo have become victims of computer attacks, resulting in interruptions in service, theft of sensitive corporate data, and vandalization of web sites.

Computer attacks are directed at all types of computer systems. The task a computer performs, or its location, can be the motivating factor in an attack. Computers connected to the Internet are often the targets because they provide services available to anyone with Internet access. Web servers, for instance, have been victimized by Denial of Service (DoS) attacks, which disrupt the availability of the web site to legitimate users. Such attacks may be directed against computers running all types of operating systems on different hardware platforms.

The Sun's Solaris Operating System [24] is widely used in universities, research laboratories, and commercial networks. Solaris is a UNIX operating system, meaning it is based on the UNIX Operating System created by AT&T Bell Laboratories. Other UNIX Operating Systems include FreeBSD, OpenBSD, and Linux. UNIX

Operating Systems run on a number of different hardware platforms. Sun's Solaris, for example, runs on both the Sparc platform and on Intel-based systems. This thesis deals exclusively with work done for UNIX Operating Systems, particularly for Solaris.

Solaris systems are frequently used as web, mail, and file servers. Because Solaris is so widely used, these systems are often the victims of computer attacks. There are many potentially dangerous attacks that target Solaris systems. The CERT® Coordination Center (CERT/CC) [3] is a federally funded reporting center for security problems. In addition to issuing security alerts and maintaining a database of known attacks, the CERT/CC ranks computer attacks according to their severity. The fifth most severe alert of all computer attacks is for a weakness in the Solaris Operating System. Any Solaris system not patched for this vulnerability is susceptible. Several other of the top 50 alerts are specific to components of the Solaris Operating System.

Intrusion Detection Systems (IDS's) are systems designed to detect, report, and possibly respond to computer attacks. IDS's analyze different types of data including network traffic, system resources, log files, and audit data. Based on the data examined, an IDS may report an attack to an administrator, create a log of the incident, modify the system configuration or firewall rules, actively respond to the threat, or all of the above.

Intrusion Detection Systems may be either host-based, or network-based. A network-based system analyzes network traffic to detect attacks, and can be setup to monitor any machine for which it can see network traffic. A host-based system monitors only the host it is installed on, and examines information such as audit data, process information, and user activity. Host-based systems are primarily used on machines running Windows, because they are single-user systems and easier to monitor. Solaris systems tend to be multi-user, and run services in which down-time is damaging. This thesis describes a non-root, host-based Intrusion Detection System for UNIX systems.

ReaLLite is a lightweight, real-time, host-based IDS. ReaLLite examines system data accessible only to programs running on a host in order to determine if an attack is taking place. The data used for this analysis includes file system data, process data, and network connection information and statistics. ReaLLite is "lightweight" enough to run in the background of any server or workstation without adversely affecting system performance.

Many Intrusion Detection Systems rely on the use of signatures to detect attack. This involves matching data examined, such as the contents of network packets, to patterns of known computer attacks. Instead of inspecting data for known patterns, ReaLLite examines the current state of the system. If there is an attack, ReaLLite can look at the effects of the attack in real-time. The attacks themselves vary by the vulnerability that they exploit, but generally perform the same actions once the host has been compromised. The majority of attack code found on the Internet will start a root shell for the intruder, create a new account on the system, or modify a critical system file. Through watching the state of the host system, ReaLLite is able to identify illegal actions that are indicative of attacks. Thus ReaLLite can identify many novel attacks that signature-based Intrusion Detection Systems might miss because they would not have a signature for it yet.

ReaLLite creates a short log file containing the warnings and messages it generates. When an attack takes place, ReaLLite creates an entry in the log file detailing the time of the attack, the type of attack attempt detected and any other relevant system information. If the attacker was successful in gaining super user privileges, or in altering restricted files, ReaLLite reports that as well.

This thesis describes the design, implementation, and testing of the ReaLLite system. It presents the results of tests, and the advantages and disadvantages of such a system design. Following this introduction, the second chapter contains background information on vulnerabilities and intrusion detection, and presents the strengths and weaknesses of several kinds of Intrusion Detection Systems. Chapter 3 focuses on the types of attacks that ReaLLite is designed to detect. Chapter 4 gives the motivation behind the design of ReaLLite. Chapter 5 presents a detailed description of ReaLLite. Chapter 6 discusses the false alarms encountered and how they were reduced. Chapter 7 contains a description of 12 attacks tested against ReaLLite, and how well it performed. Chapter 8 shows how ReaLLite was successfully ported to the Linux Operating System. Chapter 9 is a discussion of the conclusions drawn from ReaLLite's performance. Chapter 10 lists some ideas for future work.

Chapter 2

Background

A "computer attack" may deny access to the system or compromise the confidentiality or integrity of information on the system. In the most threatening type of attack, an intruder's privileges on a system are elevated. On a Windows network, these elevated privileges are usually in the form of "administrator" access. On UNIX systems, which are the focus of this thesis, these privileges are called "root" or "super user" access. A user operating at this level is capable of controlling the system on the kernel level.

An Intrusion Detection System, or IDS, is a program that tries to detect attacks against a computer system or network. An IDS may be network-based and examine network traffic to identify attacks, or may be host-based and identify attacks based on the state of a computer host. Both approaches have advantages and disadvantages stemming from their method of collecting data.

2.1 Computer Attacks and Intrusions

A computer attack is defined as "a sequence of related actions by a malicious adversary whose goal is to violate some stated or implied policy regarding appropriate use of a computer or network" [10]. This definition covers such activities as physical damage to the computer system, hijacking the domain name server registration to divert traffic, viruses, and Denial of Service attacks.

In an intrusion, an attacker gains a level of access to a system that is beyond that allowed by a security policy. The highest level of access an intruder can attain on a computer system is that of the administrator of the system. In UNIX systems this is known as root, or super user, privileges. At this level of access the user may control any aspect of the system.

There are several different kinds of computer attacks. Some examples are:

1. **Gaining physical access to the machine:** Physical access to the system could give an intruder illegal access to that system's contents, its services, or the network. This is particularly true if the machine is on a LAN with trusted hosts, if the machine does not require an account to log in from the console, or if the machine may be rebooted with an alternate boot device. Devices may also be removed and accessed directly, bypassing the computer system security completely.

2. **Exploitation of a software bug:** In this attack an intruder makes use of a bug in the operating system or an application on the system in order to gain illegal privileges. Typically, the intruder will be after the highest level of privileges on the system, that of the root user on a UNIX system. Once this level of access is granted, the intruder can launch a shell running with those privileges, allowing the intruder to execute arbitrary code.

3. **Exploiting a software misconfiguration:** An intruder might be able to make use of the improper configuration of the operating system or an application in order to gain system privileges. The system may be misconfigured, for example, to allow anonymous (or guest) logins via a telnet connection, or the permissions may be set incorrectly on critical files.

4. **Stolen or sniffed passwords:** A password sniffer is a program that passively eavesdrops on all network traffic of a shared network resource, such as on ethernet. The sniffer is capable of capturing passwords of users establishing unsecure authenticated connections on that ethernet segment.

An intruder can also gain system access or privileges through social engineering. Social engineering is an attack whereby the attacker tricks someone with legitimate access to a system into allowing the intruder access. An example of this is an intruder calling an employee of a company, and posing as a network administrator, asking for the employee's password in order to perform account maintenance.

5. **Hijacking a session:** These attacks include instances where an intruder is able to take over a previously authenticated session. A session is an established connection between two parties. An intruder may hijack a session, such as telnet, involving a machine on a shared ethernet segment by launching a Denial of Service attack against the machine on his segment and then taking over its IP address. On a computer system shared by many users, someone may be able to use the "cookies" created by web browsers of other users on the system in order to gain access to remote web sites. Many web sites store valuable personal information on their sites, using only a "cookie" to authenticate a user's identity.

While certainly not exhaustive, this list shows the wide range of intrusion attacks possible. An Intrusion Detection System might be designed to detect some, or all, of these attacks. In order to secure a computer system, however, an administrator must be ready to combat intrusion attempts of all kinds [27].

2.2 The Importance of Intrusion Detection

Intrusion Detection Systems are needed to monitor any system where access is restricted to users by a stated or implied security policy. Any computer system offering a service to users, whether networked or not, is at risk to have its security compromised. Prevention devices such as firewalls and access controls are bound to be circumvented. An Intrusion Detection System monitors one or more systems, and tries to detect and identify attacks. Attacks may be launched locally or remotely. The IDS may be able to actively respond to security threats, or may simply log anomalies and traffic that matches its signature rules.

Figure 2-1 shows a network with a shared ethernet segment. There are six machines on this ethernet, and the network is connected to the Internet. In this example, the network-based IDS monitors all traffic coming from, and destined to, the Internet. It attempts to detect attacks against all the machines on the ethernet. The host-based IDS is installed on only one host, and only detects attacks launched at that host.

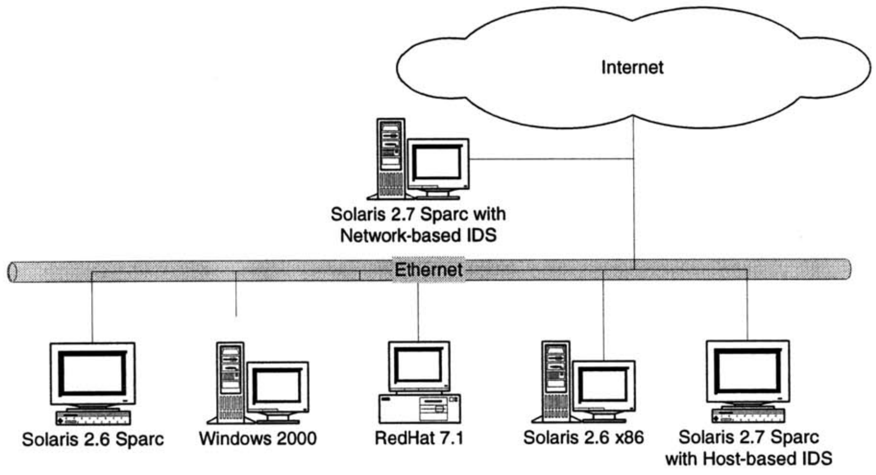

Figure 2-1: Sample network with a shared ethernet

2.3 Host-based Intrusion Detection Systems

A host-based IDS runs on one host and monitors only that host. Unlike network-based IDS's that look at network packets in search of attacks, a host-based IDS does not monitor network traffic directly. The IDS may examine file system data, system processes, network connections to and from that machine, users' activities and history of activities, and log files. A host-based system may be running at the super user level and be capable of actively responding to an attack. Such an IDS would be able to kill processes related to the attack, close network connections, and even halt the system if need be. If the IDS is running as a user process it is able to monitor and log an attack (and notify an administrator), but usually cannot stop the attack.

2.3.1 Advantages of a Host-based IDS

A host-based system has access to more data related to its host system than a network-based system does. A host-based system is able to examine the file system inode data, monitor the active processes on the system, and the users on that system. All host state information is directly visible and does not have to be inferred indirectly. Furthermore, the topology of the network does not matter. Host-based system are thus capable of detecting attacks that a network-based system cannot. One such attack is a local User-to-Root buffer overflow run from the console. A network-based IDS would not be able to detect this attack because it does not generate network traffic, but a host-based system could detect this attack.

2.3.2 Limitations of a Host-based IDS

Host-based systems can only monitor the one system that they are installed on. If many machines need to be monitored, a host-based system must be installed on each one. This may be costly if a large number of machines must be monitored and the IDS is a commercially sold product. As each of these IDS's is using system resources (cpu, memory, hard drive) it may be costly in terms of total resources to deploy IDS's on many systems. Additionally, since host-based systems usually do not examine all of the network traffic to and from the machine on which they are installed on, they do not provide a complete solution. A network-based system, on the other hand, is able to examine all packets coming to and from a machine. Host-based IDS's are best used in conjunction with a network-based IDS.

Host-based systems are becoming popular on Windows and other single-user systems. Such IDS's are more difficult to create for a multi-user operating system like UNIX. This is because Windows has security features built into the operating system that UNIX systems do not have. To get the same security features on a UNIX Operating System usually requires audit data, which is only available on a few versions of UNIX. Solaris has this capability through its the Basic Security Module (BSM). Most Solaris systems do not have BSM auditing turned on, however, so host-based

IDS's are not often used. Section 2.6 provides a description of BSM audit data and its limitations.

2.3.3 Host-based IDS's That Run As Root Are Targets

Host-based systems running as root have the advantage of being able to respond to and stop attacks in progress. They are able to do this by killing processes, logging users out, and perhaps shutting the system down. The downside of this ability, however, is that the IDS itself might be used as the target for an attack. If the IDS is setuid root, then the IDS may be vulnerable to a buffer overflow attack resulting in administrator privileges for the attacker. The IDS may be running as a root process without actually being a setuid root program, however. This can still result in a buffer overflow of the process running, or may be used by an attacker to launch a Denial of Service (DoS) attack against the host system. In a DoS attack, the fact that the IDS actively responds to a threat could be used against it. For example, if the IDS shuts down the system when a certain type of attack occurs, an attacker could launch that specific attack with the goal of shutting down the system. Although the direct attack of a buffer overflow would fail, the indirect result would be successfully denying other users access to the system.

This problem is illustrated by the UNIX ps program. The ps command runs as root, and is used to gather data concerning processes currently running on a system. It might be considered a simple host-based intrusion detection component since it can be used to determine if illegal programs, such as backdoors, are running on a system. An older version of the ps program is susceptible to a race condition vulnerability, however, and was used in attacks to gain root access to a system [10].

2.4 Review of Important Host-based Intrusion Detection Tools

There are many commercial as well as freeware Intrusion Detection Systems available. These IDS's provide a variety of different features, each with its own advantages and disadvantages. Some of these IDS's include Tripwire, CyberCop, LIDS, EMERALD eXpert-BSM, the Coroner's Toolkit, Axent's Intruder Alert, and BIDS. Reviews of these systems, and others, can be found at [1], [14], and [17]. Table 2.1 shows a comparison of these different intrusion detection programs. The following sections provide brief system descriptions, and some of their drawbacks.

Intrusion Detection System or Tool	**Real-time Intrusion Detection**	**Requires Attack Signatures**	**System runs as root**	**Kernel module needed**	**Uses BSM audit-data**	**Use File System Information**
Tripwire	No	No	Yes	No	No	Yes
CyberCop Monitor	Yes	Yes	Yes	No	Yes	No
LIDS	Yes	Yes	Yes	Yes	No	No
Coroner's Toolkit	No	No	Yes	No	No	Yes
EMERALD eXpert-BSM	Yes	Yes	Yes	No	Yes	No
Intruder Alert	Yes	Yes	Yes	No	Yes	Yes
BIDS	Yes	No	Yes	No	Yes	Yes

Table 2.1: Comparison of host-based IDS's and tools

2.4.1 Tripwire

Tripwire is a freeware and commercial host-based IDS that monitors the file system data in order to detect illegal modifications [26]. Tripwire looks at virtually all of the file system information under both UNIX and Windows, storing the information in a database along with a cryptographic hash of the actual file. Tripwire monitors a wide variety of attributes in both Windows NT and Solaris.

Windows NT	**UNIX Systems**
File adds, deletes, modifications	File adds, deletes, modifications
Flags-archive, read-only, hidden, off-line, temporary, system, director	File permissions and properties
Last access time	Access timestamp
Last write time	Modification timestamp
Create time	Inode creation/modification timestamp
File size	File type, file size
NTFS Compressed flag, NTFS Owner SID, NTFS Group SID, NTFS	Device number of the device to which the inode points. Valid only for device objects.
MS-DOS 8.3 name DACL, NTFS SACL	Number of blocks allocated
Security descriptor control and size of security descriptor for this object	user ID of owner, group ID of owner
Number of alternate data streams	Device number of the disk on which the inode associated with the file is store
Hash checking- CRC-32, MD5, SHA, and HAVAL	Hash checking- CRC-32, MD5, SHA, and HAVAL
	Growing and shrinking files
	Inode number, number of links

Table 2.2: List of data analyzed by Tripwire

After creating an initial baseline database, the Tripwire system makes subsequent checks to the file system and registry data, comparing this updated information to the baseline data. If differences in the file system are detected, the results are logged and can additionally be e-mailed to an administrator. Table 2.2 lists the different pieces of data examined by Tripwire in both Windows and UNIX Operating Systems.

Tripwire also allows the administrator to configure the system to only monitor certain attributes of files or monitor only specific files. Some files such as log files will have their contents change regularly, yet their attributes should be constant. Tripwire can be configured to take this into account and only flag changes that are unusual.

One drawback to Tripwire is that it is not a real-time system. If an attack occurs, and the system is compromised, it could be many hours before Tripwire reports the problem. Computing hashes of all the files on the system can take a great deal of time, so this process cannot be run often without seriously affecting system performance. Since it could be many hours in between an intrusion and when Tripwire verifies the integrity of files, an attacker may be able to shut down Tripwire before any alerts are issued.

2.4.2 CyberCop Monitor

CyberCop Monitor is a commercial host-based IDS for Solaris created by Network Associates [7]. This system performs both system event analysis and real-time packet analysis on traffic to and from the host. CyberCop Monitor detects attacks aimed at the host machine and those aimed at another machine in which the host is used as a "jumping off point" to launch the attack. Monitor also performs detailed logging that can be integrated with the Solaris BSM data. Monitor can create audit logs by user, event, and class, logging events down to the system call level.

CyberCop Monitor uses signatures to identify attacks, but the signatures are customizable and updatable. Administrators are able to create, or incorporate, signatures for the latest attacks into their existing IDS. Monitor also performs Event Coalescing to filter irrelevant information out of large data and log files.

Two disadvantages of CyberCop Monitor are that there is no real-time alert monitoring capability, and the IDS must run as root in order to monitor all network traffic. While Monitor has several logging options, none of them allow on-screen notification of attacks in progress. Running as root also poses potential security issues.

2.4.3 EMERALD eXpert-BSM

EMERALD's eXpert-BSM Monitor is a research host-based Intrusion Detection System for the Solaris Operating System [9]. The eXpert-BSM system performs signature-analysis in real-time, detecting "insider misuse, policy violations, privilege misuse or subversion, illegal resource manipulation, and other site policy violations." The IDS performs data collection, analysis, and provides possible solutions to security problems encountered.

EMERALD's eXpert-BSM Monitor uses audit data from the kernel to perform intrusion detection. The system also requires that eXpert-BSM be installed with root privileges in order to provide real-time analysis. Running as root could make the IDS the target of an attack. Another drawback to the eXpert-BSM Monitor is that relying solely on BSM data for intrusion detection limits detection of certain types of attacks significantly, as discussed in Section 2.6.

2.4.4 LIDS: Linux Intrusion Detection System

As the name implies, LIDS is a research Intrusion Detection System for the Linux Operating System [15]. LIDS provides security enhancements for the Linux kernel. These enhancements include: mandatory access controls (MAC), a port scan detector, file protection (even from root), and process protection. The primary feature of LIDS is its ability to limit the actions of users with root privileges. Normally, users or processes with root access are able to perform any action on other processes and files on the system. With the LIDS security enhancements, users running as root are limited by access control lists (ACLs). Both files and processes may be protected, or even hidden.

While the LIDS kernel patch can limit the powers of root, there is an inherent trust in the patch itself. Since the patch becomes part of the kernel, it is running at the highest access level. If a vulnerability were found in the LIDS kernel patch, the host system could be compromised.

2.4.5 Coroner's Toolkit

The Coroner's Toolkit is a suite of tools for UNIX used to perform forensics analysis on a system after an intrusion has occurred [5]. The suite includes information gathering tools for recovering deleted files, viewing the access patterns of files (and deleted files), and recovering cryptographic keys from a file or running process.

The Coroner's Toolkit examines the information left behind after an attack to reconstruct the details of that attack. The information examined by the suite of tools includes events from log files, the file system time stamps, examination of unknown programs that have been left behind or currently running, and examination of information hidden in and in-between files and file-systems. This after-the-fact analysis can determine how an intruder gained access to the system, what actions the intruder performed, and what files were altered.

The Coroner's Toolkit is not a real-time IDS. The suite of tools is very useful for forensic analysis of a compromised system, but does not provide any way to protect a system. The Coroner's Toolkit is only used after a system is suspected to have been broken into, and cannot offer any real-time detection.

2.4.6 Axent's Intruder Alert

Intruder Alert is a real-time, host-based commercial IDS developed by Axent Technologies [13] [17]. It is available for most commercial versions of UNIX, including Solaris. Intruder Alert performs most of its analysis on data from BSM event logs. It offers signature-based analysis for attacks, but allows for creation of new signatures and modifying the system's reaction to the attack. Intruder Alert can notify the system administrators if it detects an attack, and is also capable of taking precautionary actions to protect the system.

One disadvantage of Axent's Intruder Alert IDS is that it requires signatures to detect attacks. If Intruder Alert does not have a signature for an attack, it will not be able to detect it. Intruder Alert also relies heavily on audit data for its analysis.

2.4.7 BIDS

BIDS is a research IDS from MIT Lincoln Laboratory that relies on BSM audit data for analysis [6]. BIDS, which stands for Battlefield Intrusion Detection System, is a host-based system that analyzes audit records. BIDS attempts to detect buffer overflow attacks, unauthorized root shells, unauthorized access to user-defined system files, and the illegal creation of setuid root programs. BIDS is also capable of running in real-time. BIDS requires the use of the BSM audit data, however, and must be run as root for real-time analysis. This limits its usability to systems that support BSM auditing, and raises potential security risks associate with root processes.

2.5 Network-based IDS's

A network-based IDS analyzes network traffic to detect attacks. If the network-based IDS sees traffic to and from machines it is monitoring, the IDS will sniff that network traffic. On a switched network, the IDS might be connected to the switch and configured so that it can monitor all the machines connected to that switch. The IDS will examine the network traffic it sees, and attempts to identify patterns that look like attacks, using signature or anomaly detection, or a combinations of the two.

2.5.1 Advantages of Network-based Intrusion Detection

The main advantage of network-based intrusion detection is that fewer resources per machine monitored are needed compared to a host-based system. The network-based system also has the ability to watch all traffic to and from those machines, which cannot be done by a host-based system. A host-based system needs super user privileges on a machine in order to put the ethernet card into promiscuous mode. The machines being monitored by a network-based IDS do not need any program installed or process running. Since the analysis is all passive, the IDS should not hinder the performance of the network. A network-based system can also readily identify attacks or probes against multiple machines.

2.5.2 Limitations of Network-based IDS's

The disadvantage of a network-based IDS is that it is very difficult to determine exactly what is happening on a particular machine by watching only the network traffic. A network-based IDS might have many machines to watch, all possibly running different operating system, services, and even versions of those services. With only the data in the network packets to examine, it is impossible for the IDS to determine or keep track of the exact state of all the machines. To make matters worse, the IDS does not have any guarantee that the machine it is monitoring will see the same packets that it sees. As described in [19] and [11], there are several kinds of attacks that take advantage of weaknesses in network-based IDS's. They include insertion, evasion, and de-synchronization. These three weaknesses and the always imminent use of encryption as an IP standard severely limits the utility of network-based system.

The first type of attack that can evade a network-based IDS is an insertion attack. An insertion attack requires an IDS that does not calculate checksums for packets nor keep track of the sequence numbers. In this attack an intruder inserts packets with incorrect checksums or invalid sequence numbers into the stream of data. The destination system will discard the packets as being bad, but the IDS treats them as if they were received and processed correctly. By inserting a FIN or RST packet, the intruder can trick the IDS into thinking the connection has been closed. That attacker might also insert spoofed data in the network traffic that will disrupt an IDS searching for particular strings.

The second type of attack that can evade a network-based IDS is an evasion attack. In an evasion attack, the IDS sees all of the packets that the end-system does, but misses the actual attack. This can be accomplished by fragmenting the IP packets that contain the attack. Most IDS's cannot properly handle all the different ways an IP packet can be fragmented, and thus cannot detect attacks contained within fragmented packets. For example, an intruder might scramble the order of the fragmented IP packets, or overlap the positions of some packets. Unless the IDS can handle the fragments in the exact same manner as the end-system, it will end up constructing a different packet and miss the attack.

The final type of attack that can evade a network-based IDS is a de-synchronization attack. A de-synchronization attack tries to fool the IDS into monitoring a fake connection to the end-system instead of the real one. This attack requires the IDS to keep track of sequence numbers. The attack is accomplished by sending a post connection SYN packet in the data stream, for a connection already established. This post connection SYN packet will have a different sequence number, marking it as a different connection. The IDS will resynchronize the sequence numbers, but the end-system will discard the second SYN packet. This results in the IDS missing the real traffic to the end-system. If the IDS ignores post connection SYN packets, the same effect can be accomplished by sending a pre-connection SYN with an invalid checksum, followed by the legitimate SYN packet. The IDS will treat the first SYN as legitimate and discard the second, while the end-system will do the opposite.

The only way to fully avoid these three types of attacks and to perform intrusion detection when network traffic is encrypted is to use a host-based system where the host's status can be accurately determined.

2.6 BSM Auditing

The Basic Security Module (BSM) of the Solaris Operating System is a kernel option that logs hundreds of different system events. These events generate records consisting of different types of tokens. Depending on the event, the tokens contain different pieces of information. Users that log into a system are assigned a unique identification number, called an auditID. This auditID is then inherited by all processes that the user creates, even if su is used to change identities. The auditID information makes identifying who caused a specific event very straightforward, and a user's actions can be easily tracked through the subsequent events logged. As discussed earlier in this chapter, BIDS, CyberCop Monitor, EMERALD eXpert-BSM, and Intruder Alert are host-based IDS's for Solaris that use BSM auditing.

2.6.1 Limitations of BSM Auditing

The most important limitation of BSM auditing is that it is available only under the Sun Solaris Operating System. No other UNIX Operating System generates BSM audit data. This means that any tools or IDS's that incorporate BSM audit data will only work for Solaris systems, so not many are developed.

Auditing all kernel events with BSM results in a huge volume of low-level data collected. Since no BSM tool comes standard with the Solaris Operating System to automate the process of examining the event stream, either a commercial tool has to be used or it must be done by hand. This type of analysis by hand would be very time consuming and difficult. BSM events need to be carefully filtered to allow only critical events.

While BSM audit data provides detailed records of events, it is not an ideal source of intrusion detection input data. The events recorded give a good picture of the system calls being made, but they do not tell much about the higher-level state of the system. In fact, BSM misses important information. There is no BSM information regarding cpu or memory usage, which may be indicative of some Denial of Service attacks. BSM audit data also does not contain information regarding the contents of files on the system. If an attack added a user entry in the password file, for instance, an IDS using BSM audit data would not be able to locate that new entry. Probes and scans are difficult to detect as well, since BSM data does not contain information about network packets.

BSM auditing may also allow a host to be the victim of a Denial of Service attack. If an attacker knows BSM auditing is turned on, and which events are logged, it is possible to create many BSM audit data records through repeatedly performing events that are logged. This floods the system with audit data, making further analysis and detection of a real attack difficult or even impossible.

Another limitation is that most systems do not turn BSM auditing on. It is not activated by default when Solaris is installed, and most users never activate it.

Chapter 3

Attack Types

Computer attacks can be categorized as affecting the availability of a host or service, or compromising the confidentiality, integrity, or the security protection of a system [12]. Some examples of vulnerabilities that computer attacks take advantage of include input validation errors, boundary condition errors, buffer overflows, configuration errors, race conditions, and design errors. Attacks can also be categorized as being launched locally or remotely.

The ReaLLite Intrusion Detection System described in this thesis focuses on buffer overflow, Denial of Service, and probe attacks. These attacks and their descriptions are summarized in Table 3.1. Buffer overflow attacks are the most severe type of attack, with the potential for an attacker to gain root access to a system. Denial of Service attacks can cause costly down-time for servers, and are becoming a common attack, especially against web servers. Denial of Service attacks are relatively easy to execute, but hard to prevent. While not as threatening by themselves, probe attacks are often the first step in a larger, more threatening attack.

According to The ICAT Metabase from the Computer Security Division at the National Institute of Standards and Technology [12], there are 210 alerts for known vulnerabilities in the Solaris Operating System. Seventy-four, or 35% of all alerts are for buffer overflow vulnerabilities. A buffer overflow attack takes advantage of a programming error in a specific executable file, and may allow a remote or local user to gain root privileges on the victim machine. Sixty-one of the 74 buffer overflow vulnerabilities were categorized as high severity.

Denial of Service attack are also very common. In this type of attack, legitimate users are denied access to the system by the attacker. Several high-profile web sites such as eBay, Microsoft, and Yahoo have recently been victims of this type of attack. According to the ICAT Metabase, there were 31 known vulnerabilities that resulted in denying availability to users. This number represents 15% of all known Solaris vulnerabilities. Seven of these Denial of Service attacks were categorized as high severity.

Another type of attack is a probe. A probe gathers information about a system for use in future attacks, such as a buffer overflow attack. A probe attack might consist of locating all systems on a network, determining which services a computer runs, or identifying the operating system of a system.

Attack Type	Attack Description
Buffer Overflow (U2R)	A local user is able to gain "root" privileges on a UNIX system through a buffer overflow.
Buffer Overflow (R2L)	A user is able to gain local access to the system through performing a buffer overflow attack on a system daemon listening for incoming connections. The attack is launched from a remote machine connected only through a network connection.
Denial of Service	Legitimate users are denied access to the system or a service provided by that system.
Probe	Reveals information about a system, such as operating system and ports open

Table 3.1: Type of attacks used for testing ReaLLite

3.1 Buffer Overflows

Some buffer overflow attacks take advantage of improper bounds checking on an argument or other data that a program is given [23]. The actual code being exploited copies a buffer supplied by the user into a data structure defined by the code without checking to see whether the given data fits into that structure. A problem arises when the data supplied by the user is larger than the size of the structure. When this happens, the data "overflows" the structure set aside for it, and overwrites part of the memory stack. If the return address on the stack of the calling function is overwritten, arbitrary code can be executed. This code is executed with the privileges of the program that had its stack rewritten. If this program was setuid root, then the arbitrary code is executed with super user status. An intruder can use this to his advantage to gain system privileges.

Another type of buffer overflow can occur in programs that use data from the user as a formatting parameter for an output routine [4]. The family of output routines includes the printf(), sprintf(), and snprintf() functions. If a program allows the user to specify the output format from one of these functions, an attacker can take advantage of this and perform a buffer overflow attack. This can overrun the buffer and allow the attacker to have complete control over the program.

Buffer overflow attacks may take place locally on a machine, or be launched via a network connection. Processes that listen for TCP/IP connections may have their stack overrun if they perform improper bound checking on arguments passed over the network connection. Most remote buffer overflow attacks do not require that the attacker have an account on the victim machine. These buffer overflow attacks can also be part of a larger attack. For example, the Solaris sadmind vulnerability [20] was used as part of a major Distributed Denial of Service (DDoS) attack recently.

3.1.1 Local Buffer Overflows (U2R)

In local buffer overflow attacks, the user launching the attack already has access to the system. The user may have obtained access legally or may have illegally acquired local access through another attack such as social engineering. The attacker may use a buffer overflow exploit to gain additional system privileges, including root access. Often these types of attacks will result in the attacker obtaining a "root shell." A root shell is a shell process, such as /bin/sh or /usr/bin/tcsh, which is running with the effective user ID of the super user root. These attacks are known as "User to Root" or "U2R" attacks [8].

3.1.2 Remotely Executed Buffer Overflows (R2L)

In a remotely executed version of a buffer overflow attack, the attack is launched via a network connection and provides a remote attacker with root or user access to the system. This access is usually in the form of a root shell, similar to a local buffer overflow attack. This type of attack is most effective when launched against a daemon process running as root, and allowing anyone to establish a connection with it. Examples of such daemons include named (DNS daemon), ftpd, httpd, and fingerd. Buffer overflow attacks that allow local access through such daemons are known as "Remote to Local" or "R2L" attacks [8].

These attacks might also be part of an network "worm". A worm is a self-replicating program that gains illegal access to a computer system. A worm might use a buffer overflow to gain access to a system, then use that system as a launching point to attack other systems in the same manner.

3.2 Denial of Service Attacks

The goal of a Denial of Service, or DoS, attack is to disrupt the legitimate use of a system by other users. DoS attacks can take many forms such as DNS record hijacking, halting a Windows NT machine with the Blue-Screen-of-Death, or flooding a network with large numbers of bogus requests. The integrity of the victim machine is never compromised in a DoS attack. No files are altered, or additional privileges gained. The attacker simply denies others access to some service of the victim machine. Two common types of DoS attacks SYN Floods, and resource exhaustion. A SYN Flood attempts to stop other users from accessing a service over a network. In a resource exhaustion attack, the attacker attempts to use up all of the resources of the victim's machine so that other users cannot access it. These attacks are described in more detail in the following paragraphs.

3.2.1 SYN Floods

A SYN Flood [10] is an example of a DoS attack. SYN floods take advantage of an inherit weakness in the TCP protocol and have become a common attack against web sites. TCP connections are established using a three-way handshake [25]. This handshake consists of the initiating system sending a SYN packet, the destination system responding with a SYN-ACK packet, and the initiating system replying to the SYN-ACK by sending an ACK packet. Figure 3-1 shows the three stages of establishing a connection. In this example, A is initiating a connection with B.

$$\begin{tikzcd}
A \arrow[r, "\text{SYN}"] & B \\
A & B \arrow[l, "\text{SYN/ACK}"'] \\
A \arrow[r, "\text{ACK}"] & B
\end{tikzcd}$$

Figure 3-1: Three stage TCP/IP handshake

All three steps are needed to establish a TCP connection. In a SYN Flood attack, the system launching the attack will "spoof" a large number of SYN requests destined to the victim system. These requests may appear to be coming from anywhere. The victim responds with SYN-ACK packets for each SYN request it receives. The attacker never responds to these SYN-ACK packets, and most likely does not even see them if the original SYN's had spoofed source IP addresses. The TCP connection is thus never fully established because the last part of the three-way handshake never occurs. The victim machine thus waits for the ACK packet, which never comes. Eventually the victim machine will give up and stop listening for the packet. At any given time, however, the victim can only be listening for a finite number of connections. If the attacker can fill the victim's connection queue with enough half-open connections, the victim will be unable to listen for any other connections. This results in legitimate users being unable to connect to the system.

3.2.2 Resource Exhaustion

In a resource exhaustion attack, the goal is to render the system unusable for other users by exhausting system resources such as file space, memory, processes, or network connections. If all the file space on a partition is filled, then processes cannot write any data to disk, including any temporary files that many programs need in order to run. Much the same way, if all the system memory (and swap space) are used by processes, then new processes cannot allocate new memory and existing processes cannot increase the memory that they use. Similar attacks can exploit the limits on the total number of processes, and the number of simultaneous network connections.

3.3 Probe

A probe is an attack that gathers information about a remote system or network. This information can later be used for additional attacks such as Denial of Service attacks, or intrusion attempts. Often the first step for an attacker is to gather as much information as possible about a network or system. This information might include IP addresses that respond to TCP/IP packets, what operating system a system is running, and what services a system offers. Probe attacks are used to collect this type of information. Many programs, such as nmap [18], are capable of performing ping sweeps, operating system fingerprinting, and port scans, which are common probes. These probes are described in detail in the next paragraphs.

3.3.1 Ping Sweep

A ping sweep locates the IP addresses that respond to TCP/IP packets. A "ping" is an "echo message" ICMP packet used to determine if a machine is responding to network packets, or "alive." If the target of the ping packet is alive, it will respond with an "echo reply message" ICMP packet. A ping sweep sends these echo messages to many IP addresses, and listens to the responses. For example, an attacker might perform a ping sweep on MIT's network by sending ping packets to all IP addresses matching 18.*.*.*. This would provide the attacker with a list of alive hosts at MIT.

3.3.2 Operating System Identification

A TCP/IP Operating System fingerprinting program can determine a machine's operating system quite accurately by listening to the victim's responses to TCP packets that are not defined in the TCP specifications. Since there is no official response to these packets, different operating system vendors opted to respond to them in different ways. The differences across platforms are significant enough to identify the victim's operating system down to the version of the kernel for a Linux system. The advantage of this is clear: identifying a remote machine's operating system is the first step in almost every direct attack on a system, such as a buffer overflow. By detecting the operating system of a computer, an attacker can then start to look for specific holes and bugs for that specific operating system.

3.3.3 Port scan

A port scan is an information gathering attack used by hackers before launching additional attacks, such as a buffer overflow. A port scan consists of the attacker connecting to various ports of the victim machine in order to determine what services are currently running (i.e. web server, dns server, ftp server). A port scan might include attempting to connect to all ports from 0-65535, or may be as few as a couple ports. The purpose of a port scan is to provide the attacker with information about which servers are running on a system so that weaknesses in those servers might be later exploited.

Chapter 4

Design Approach

The motivation for creating the ReaLLite system was to test how reliably attacks could be detected in real-time by a lightweight system running as a user process, without the use of kernel modules or BSM audit data. ReaLLite was developed for Solaris, but designed to use standard UNIX programs, so that porting to any UNIX platform is possible. The system was also designed to be easy to use and configure, and can be run on any workstation or server. Figure 4.1 lists the desired characteristics in designing the ReaLLite system, and the advantages of such a design. The rest of this chapter presents a more detailed description of these characteristics, and the reasons for choosing them.

4.1 Lightweight

A main design goal was to make ReaLLite lightweight. A lightweight system is able to run in the background of any workstation or server with low enough cpu and memory usage so as not to detract from the system's other jobs. In fact, a user on the system should not be able to tell that ReaLLite is running, judging from the performance and responsiveness of the host. The IDS should also take up a minimal amount of hard drive space, though this was not a main concern as hard drive space is relatively inexpensive.

Desired Characteristics	Advantage
Lightweight	Can be easily deployed on any system. Low disk and cpu usage, so it does not effect system performance.
Real-time	Detection of attacks as they occur.
Host-based	Access to a greater amount of data about the host system.
Uses Standard Utilities	Can use other setuid root programs to gather the information needed.
No BSM Audit Data Needed	Can run on other UNIX systems besides Solaris.
No Kernel Module Required	No special modification to system needed. System does not need direct access to kernel layer information.
Runs As a User Process	IDS does not pose security concerns associated with setuid root programs, and is easy to install.
Portable	Nothing about the system is specific to one operating system. Capable of being ported to other UNIX systems.
Uses File System, Process, Network, and System Resources Information	Can detect firsthand the effects of an attack on the system.
Non Signature-based Detection	Signatures are not needed to detect attacks. Provides better detection of novel attacks.

Table 4.1: Desired characteristics of ReaLLite

4.2 Real-time Attack Detection

A real-time system is able to detect an attack while it is running, or in a short time period after it completes. In contrast, some Intrusion Detection Systems only check periodically through log files looking for malicious activity. These systems may not discover attacks until hours or days after they might have occurred. ReaLLite was designed to identify attacks as they happen.

4.3 Host-based Approach

The initial goal of this system was to detect buffer overflow attacks. These attacks are the most severe, and are very hard to detect by inspecting network traffic unless a signature exists for the particular attack. Even with a signature, these attacks can circumvent a network-based IDS as discussed in Section 2.5.2. Signature or rule based checking will also not catch novel buffer overflow attacks. A host-based system has the ability to examine the current state of both the host system and its files, thereby making detection of these attacks easier.

4.4 Use of Standard UNIX Utilities

Standard UNIX utilities are used to collect the data that the IDS uses to detect attacks. There are two main reasons for this: (1) these UNIX utilities are common and fairly uniform across platforms, making the porting of ReaLLite to another operating system simpler, and (2) using existing tools that are setuid root allows ReaLLite to run as a user process. In Solaris, for example, a user does not have direct access to information in the /proc directory concerning processes that he does not own. The result of this is that an IDS must either run as root, or use a setuid root program like "ps" to access information for all processes currently running. Table 4.2 lists the stardard UNIX sources of information used by ReaLLite and the data gathered from each.

To help examine the state of the host, "netstat" is used to gather network data, and "ps" for process data. These commands are called with various arguments in order to collect the needed information. Standard UNIX commands like awk, grep, and wc are used to format the output for easier parsing by ReaLLite.

Standard UNIX header files are used to access the inode and user information data. The same information could have been gathered using the "ls" and "w" commands with different arguments, and parsing their outputs. This would have been slower and created additional processes, so the same functionality was built directly into

ReaLLite.

UNIX Information Source	Data Gathered
ps	Active process information. Data includes real and effective user ID's of the owner, the Process ID (PID) number, the Parent PID, how much memory and CPU the process is using, the name of the command executed, and the name of the program running
netstat	Network statistics and connection information. Data list of active TCP/IP connections, SYN packets received, and outbound TCP Reset packets sent
find	Used to find setuid root and critical files on a UNIX system
awk, grep, wc	Programs used to format and parse the information from other UNIX utilities like netstat
inode data	Inode information includes last access time, last modification time, and last inode data status change times. Information is accessed through the stat() function.

Table 4.2: UNIX information analyzed by ReaLLite

4.5 BSM Audit Data Not Needed

The system was also designed with the idea that it would not need BSM auditing data. BSM audit data requires special kernel settings, and an additional utility to monitor the audit data and locate attacks. This is not easily deployable and the use of BSM auditing is rare. In addition, the use of BSM audits would make it impossible to port ReaLLite to other UNIX Operating Systems.

4.6 ReaLLite Runs as a User Process

Running ReaLLite as the super user, root, would allow the IDS to be the target of an attack. An intruder might try to launch a buffer overflow attack against ReaLLite or use the IDS to crash the host system. Allowing the system to run as root increases ReaLLite's power to halt an attack in progress, but also makes the IDS a potential security risk. Some administrators might also be wary of running a setuid root IDS on their networks. Running the system as a normal user process means that the system is made no less secure by running it. It also makes ReaLLite easier to install on a system since you do not need root access.

Future version of ReaLLite may run components of the system as root. This is a tradeoff as more information is then available to the IDS, but at the cost of a potential security problem.

4.7 Portability

ReaLLite was designed to be portable across UNIX Operating Systems. It is built around standard UNIX utilities and uses UNIX inode data. Solaris was used as the host operating system for the proof-of-concept IDS because it supports all of the features and utilities needed and is widely used. Porting to other platforms is possible, though, and is demonstrated further in Chapter 8.

4.8 Proof-of-Concept System

The current system follows these design specifications, but is not meant to be a final product. The code developed is more of a proof-of-concept product, showing that it is possible to detect a large number of attacks with the types of data available to a user. It is not intended for commercial use in its present form.

Chapter 5

Details of the ReaLLite IDS

Figure 5-1 shows a block diagram of the ReaLLite system. The diagram breaks down ReaLLite into two separate parts: the "low-level" and "high-level" scans for buffer overflow attacks on the left, and Denial of Service and probe attack detection on the right side. Alerts may be issued at any stage based on the information from the host system.

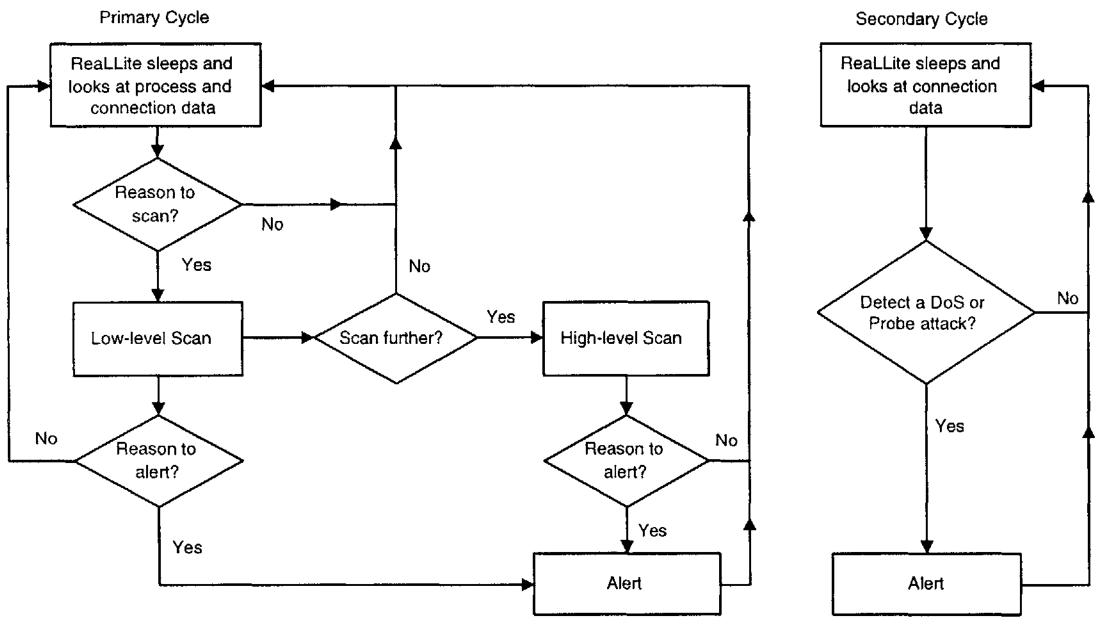

Figure 5-1: A block diagram of ReaLLite system

The ReaLLite IDS examines system data of the host machine through standard

UNIX programs like ps and netstat, file system inode data, and system log files. As shown in Figure 5-1, ReaLLite can viewed as divided into two separate detection cycles. These cycles are running continuously in the background, using few system resources.

The left side of Figure 5-1 shows how ReaLLite scans process, connection, and inode data. This is the primary cycle of the ReaLLite system, and occurs semi-periodically with a random delay of zero to ten seconds between successive cycles. If ReaLLite detects during one of these cycles that the state of the system has changed, a low-level scan is performed. A low-level scan triggers a high-level scan if setuid root programs have been accessed. This high-level scan takes a much more detailed look at the inode data and log files of the system. Based on the information from the low and high level scans, ReaLLite tries to determine if a buffer overflow attack took place.

The right hand side of Figure 5-1 shows how ReaLLite attempts to detect DoS and probe attacks through the network information gathered from netstat. This is the secondary cycle and occurs once for every three traversals of the primary cycle.

The rest of this chapter is devoted to explaining the details of the ReaLLite system. The following sections discuss the system resource usage of the IDS, the evidence examined by the system, the two levels of scanning, and the order in which the cycles are executed.

5.1 ReaLLite System Resource Usage

ReaLLite is a lightweight system, meaning that it does not consume many system resources. As seen in Figure 5-2, the total size of the ReaLLite source code, executable file, and all dependencies is under 120 Kilobytes. Typical hard drives are measured in Gigabytes, making the size of ReaLLite insignificant. While running, ReaLLite uses less than 1% of the CPU time, and around 1 Megabyte of RAM. Figure 5-3 is the output of the "top" program, which shows ReaLLite's CPU and memory usage. The average CPU usage over a few days running on a server was less than 0.2%. While this does not include the CPU usage of the separate ps and netstat processes that were spawned, the overall CPU usage of ReaLLite is minimal, and does not affect the performance of other processes and services on the host.

```
debris:~/code> ls -l
total 224
drwxrwxrwx   2 kem      staff        512 May 14 00:58 ./
drwx------   4 kem      staff       1024 May 13 22:54 ../
-rw-rw-rw-   1 kem      staff       7421 May 13 22:41 allowedfiles.lst
-rw-rw-rw-   1 kem      staff       8975 May 13 22:49 criticalfiles
-rw-rw-rw-   1 kem      staff       1034 May 13 22:41 rLstructs.h
-rwxrwxrwx   1 kem      staff      45904 May 13 22:54 reaLLite*
-rw-rw-rw-   1 kem      staff      44771 May 13 22:54 reaLLite.c
-rw-rw-rw-   1 kem      staff       1170 May 13 22:50 suidlist
debris:~/code> du -k
111     .
debris:~/code>
```

Figure 5-2: ReaLLite harddisk usage

5.2 Evidence Examined By the System

Since the ReaLLite IDS was designed to run as a user process on the Solaris Operating System, data such as process data for the entire system cannot be examined without the use of a setuid root program. This process data is stored in the /proc directory on most UNIX's, but Solaris only allows a user to read data in /proc concerning owned processes. The "ps" program, however, is setuid root on Solaris and provides ReaLLite with the information needed regarding all users' processes. The "netstat" program provides network information, and ReaLLite directly reads both the system inode data and system log files to get the the rest of the information that needs monitoring. The following sections explain what data ReaLLite extracts and analyzes from the file system, ps, netstat, and UNIX log files.

```
load averages:  0.20,  0.14,  0.18
01:10:53
35 processes:  33 sleeping, 1 zombie, 1 on cpu
CPU states: 98.0% idle,  0.0% user,  2.0% kernel,  0.0% iowait,  0.0% swap
Memory: 256M real, 208M free, 7792K swap in use, 808M swap free

  PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
 2011 kem        1  33    0 1200K 1040K cpu     0:03  2.09% top-5.6
  189 root       9  33    0 2416K 1896K sleep   0:01  0.13% nscd
 1999 kem        1  33    0 1008K  920K sleep   0:00  0.09% reaLLite
    1 root       1  33    0  696K  208K sleep   0:14  0.05% init
 1983 kem        1  19    0 1984K 1848K sleep   0:00  0.01% tcsh
  165 root       4  33    0 3872K 2688K sleep   0:51  0.00% automountd
  297 root       1  34   -2 2648K 1568K sleep   0:04  0.00% customs
  182 root       1  23    0 1720K 1344K sleep   0:02  0.00% cron
  112 root       1  33    0 2088K 1264K sleep   0:02  0.00% rpcbind
  229 root       5  15    0 2336K 1168K sleep   0:01  0.00% vold
  114 root       4 -20    0 2008K  968K sleep   0:00  0.00% keyserv
  308 root       4  11    0 2648K 1160K sleep   0:00  0.00% dmispd
 1981 root       1  13    0 1560K 1272K sleep   0:00  0.00% in.rlogind
  321 root       1  17    0 5760K 1344K sleep   0:00  0.00% dtlogin
  149 root       1  18    0 1768K  912K sleep   0:00  0.00% lockd
```

Figure 5-3: ReaLLite CPU and memory usage

5.2.1 File System Data

In a UNIX Operating System, such as Solaris, executable files can have a "setuid" bit set. This setuid bit allows the file to execute with the privileges of the owner of the file, not of the user actually running the program. It changes the user ID of the process to the owner of the file. For example, on most UNIX systems the program "ping" is owned by root and has its setuid bit set. The ping program will execute with the privileges of root regardless of who starts the process.

An inode, or index node, is a data structure that contains information about files stored in the file system. In UNIX systems this information includes time stamps. The inode contains time stamps for the time that the file was last accessed, the time it was last modified, and the time in which the inode data was last modified. Note that accessing the inode data of a file does not change the last access time of that file. The inode data is not considered part of the file data.

ReaLLite examines the inode data directly using standard UNIX system calls to access the information in the "stat" data structure. Figure 5-6 shows this data structure. The UNIX command "ls" could have been used to access this data, but would have been slower and required spawning many more processes.

ReaLLite uses information from the file system to aid in the detection of attacks. The main pieces of information used from the file system are: the names and locations of all setuid root programs, the last access time from the inode data for these setuid root programs, and the last modified and inode status change times of critical files on the system. A critical file is one used by the operating system to enforce a security policy. Examples include the /etc/shadow and /etc/passwd files, and other files that should not normally be accessed by specific users.

Location of All SetUID Root Programs

Setuid root programs are the primary targets of buffer overflow attacks. If a list of setuid root programs is not provided, ReaLLite generates a list of all the setuid root programs on the system. This list is generated through the "find" command, another standard UNIX program. Figure 5-4 demonstrates how this file is generated, and the contents of a sample suidlist file. The IDS subsequently monitors the last access times of all the files in this suidlist. A typical Solaris 2.X system has around 100 setuid root programs.

List of Critical System Files

ReaLLite is given a list of files on the host that are critical to the security of the system. This list of files typically includes everything in the /etc directory and any other file of the system that the administrator believes needs to be protected. Because ReaLLite must be able to read the inode data for these files, they must be in a directory that the user owning the ReaLLite process can read. If no critical filelist is provided, ReaLLite generates a generic one using the "find" command.

Figure 5-5 shows the command used to generate this list, and a sample file. The inode data for files in this list is subsequently monitored for recent changes.

```
debris:~> find / -perm -4000 -user root > suidlist
debris:~> more suidlist
/usr/lib/lp/bin/netpr
/usr/lib/sendmail
/usr/openwin/bin/xlock
/usr/bin/at
/usr/bin/crontab
/usr/bin/eject
/usr/bin/fdformat
/usr/bin/login
/usr/bin/passwd
/usr/bin/rcp
/usr/bin/rlogin
/usr/bin/rsh
/usr/bin/su
/usr/bin/uptime
/usr/bin/w
/usr/bin/lpset
/usr/sbin/ping
```

Figure 5-4: Command to generate setuid list and sample file

```
debris:~> find /etc > criticalfiles
debris:~> more criticalfiles
/etc/cron
/etc/ff
/etc/format
/etc/fs
/etc/fsck
/etc/getty
/etc/group
/etc/halt
/etc/hosts
/etc/inet
/etc/mkfs
/etc/mount
/etc/mountall
/etc/net
/etc/passwd
/etc/resolv.conf
/etc/shadow
/etc/umountall
/etc/wall
/etc/whodo
```

Figure 5-5: Command to generate critical file list and sample file

Inode Data From the File System

Figure 5-6 shows the data structure used to access the inode information stored in the file system. The three pieces of inode information used by ReaLLite are: the last access time, the last modification time, and the file status change time.

1. **Last access time:** This time stamp is updated to the current time of the system when a user or process runs the program, or accesses the data stored in the file. Accessing the inode data does not update the last access time for the file.

2. **Last modification time:** This time stamp is updated to the current time of the system when the contents of the file are changed.

3. **File status change:** This time stamp is updated to the current time of the system when the inode data for the file is modified. Some attacks change the owner or permissions of a file. This information is stored in the inode data, so any such change will update this time stamp.

```
mode_t    st_mode;       /* File mode (see mknod(2)) */
ino_t     st_ino;        /* Inode number */
dev_t     st_dev;        /* ID of device containing */
                         /* a directory entry for this file */
dev_t     st_rdev;       /* ID of device */
                         /* This entry is defined only for */
                         /* char special or block special files */
nlink_t   st_nlink;      /* Number of links */
uid_t     st_uid;        /* User ID of the file's owner */
gid_t     st_gid;        /* Group ID of the file's group */
off_t     st_size;       /* File size in bytes */
time_t    st_atime;      /* Time of last access */
time_t    st_mtime;      /* Time of last data modification */
time_t    st_ctime;      /* Time of last file status change */
                         /* Times measured in seconds since */
                         /* 00:00:00 UTC, Jan. 1, 1970 */
long      st_blksize;    /* Preferred I/O block size */
blkcnt_t  st_blocks;     /* Number of 512 byte blocks allocated */
```

Figure 5-6: Data structure for file inode data

5.2.2 System Process Data

ReaLLite needs to monitor processes on the system, but not all processes. Processes that are neither running as root (as in a setuid root program) nor owned by root cannot modify any of our critical files, so the IDS does not keep track of those. Only processes with a real or effective user ID of root are monitored. Figure 5-7 shows the output of the ps command that ReaLLite invokes to gather process data. This output is then parsed by the IDS and the data is stored in memory. The two types of processes that ReaLLite identifies and keeps track of are processes running as root, and processes running as setuid root.

List of Root Processes Running

These are the processes that are running with real and effective user ID's of root. These processes include any programs that the root user may have started, as well as system daemons that were started by the init scripts on start up.

List of SetUID Root Programs Running

Setuid root processes may have been started by any user on the system and will have an effective user ID of root, and a real user ID of the user who started the process. Examples of standard setuid root programs on a system include "w", "passwd", "ping", and "ssh."

5.2.3 Netstat Data

Netstat displays network connections and interface statistics, among other information. By passing certain parameters to netstat and parsing the output, ReaLLite is able to keep up-to-date information on certain network activity. The three main pieces of information obtained from netstat are: a list of established TCP connections, a count of the number of TCP reset (RST) packets sent out, and the number of SYN packets received that have not been established into connections or timed

```
debris:~> ps -ef -o ruser,user,pid,ppid,time,pcpu,vsz,fname,comm
   RUSER     USER   PID  PPID        TIME %CPU  VSZ COMMAND  COMMAND
    root     root     0     0        0:01  0.0    0 sched    sched
    root     root     1     0        3:57  0.0  644 init     /etc/init
    root     root     2     0        0:00  0.0    0 pageout  pageout
    root     root     3     0    06:46:24  0.5    0 fsflush  fsflush
    root     root   164     1        0:09  0.0 2768 syslogd  /usr/sbin/syslogd
    root     root   138     1        0:00  0.0 1440 inetd    /usr/sbin/inetd
    root     root   111     1        0:01  0.0 1956 rpcbind  /usr/sbin/rpcbind
    root     root   113     1        0:00  0.0 1848 keyserv  /usr/sbin/keyserv
    root     root   160     1        0:25  0.0 3640 automoun /usr/lib/autofs/automountd
    root     root   215     1        0:36  0.0 1796 sendmail /usr/lib/sendmail
    root     root   173     1        1:34  0.0 1456 cron     /usr/sbin/cron
    root     root 24386    65        7:03  0.0 1784 sshd1    /usr/local/sbin/sshd
    root     root   225     1        0:02  0.0  852 utmpd    /usr/lib/utmpd
    root     root   294     1        0:00  0.0 1472 ttymon   /usr/lib/saf/ttymon
     kem      kem 28765 28762        0:02  0.0 1920 tcsh     -tcsh
    root     root 28762    65        0:05  0.0 1732 sshd1    /usr/local/sbin/sshd
    root     root   297   293        0:00  0.0 1484 ttymon   /usr/lib/saf/ttymon
    root     root 28742    65        0:05  0.9 1732 sshd1    /usr/local/sbin/sshd
     kem     root   172 28745        0:00  1.4  800 ps       ps
debris:~>
```

Figure 5-7: Output from ps command used by ReaLLite

out. The following paragraphs explain these three pieces of information, and how ReaLLite extracts them from the netstat command.

List of Active TCP Connections

ReaLLite keeps track of all connections to the machine that it is running on. This connection data includes the source IP addresses and ports of all connections, and their destination address (the host machine) and port number. Figure 5-8 shows the output of the command used by ReaLLite. This list shows connections to the NFS daemon on port 2049, a ssh connection on port 22, and an imap connection on port 143, as well as various other established TCP connections.

Count of TCP Resets Sent

According to RFC 793 [25], closed ports must respond to packets by sending a TCP reset (RST). Open ports will either try to establish a connection, or discard the packet depending on its status. Figure 5-9 shows the command used by ReaLLite to get a count of TCP RST packets sent out by the system. It shows that there were 3,315 outbound TCP reset packets sent out since the machine last rebooted.

```
saturn% netstat -n -P tcp | egrep 'SYN|ESTAB|ACK' | awk '{print $1, $2, $7}'
192.5.135.115.2049 192.5.135.43.1023 ESTABLISHED
192.5.135.115.2049 192.5.135.134.1022 ESTABLISHED
192.5.135.115.2049 192.5.135.54.1022 ESTABLISHED
192.5.135.115.2049 192.5.135.123.1022 ESTABLISHED
192.5.135.115.42590 192.5.135.4.32864 ESTABLISHED
192.5.135.115.52600 192.5.135.14.10000 ESTABLISHED
192.5.135.115.22 192.5.135.10.32837 ESTABLISHED
192.5.135.115.143 192.5.135.60.38500 ESTABLISHED
192.5.135.115.2049 192.5.135.4.1023 ESTABLISHED
192.5.135.115.42590 192.5.135.4.32842 ESTABLISHED
192.5.135.115.53357 192.5.135.14.10000 ESTABLISHED
192.5.135.115.53360 192.5.135.115.936 ESTABLISHED
192.5.135.115.936 192.5.135.115.53360 ESTABLISHED
saturn%
```

Figure 5-8: Netstat command to generate a list of all established TCP/IP connections

```
debris:~> netstat -s -P tcp | grep tcpOutRsts | awk '{print $3}'
3315
debris:~>
```

Figure 5-9: Command to get a count of outbound TCP resets

Count of SYN Packets

A three-way handshake is needed to establish a TCP connection [25]. In this handshake the machine initiating the connection sends a SYN packets to the receiving host. The receiving host responds with a SYN-ACK packet, to which the initiating machine answers with an ACK.

The netstat command show in Figure 5-10 displays the number of connections in between the second and third stages of the three-way handshake. This figure shows that there were no SYN packets received that had not been fully established or dropped yet. This number represents the number of connections waiting for the SYN-ACK packet. After a certain period of time, the host system will give up on establishing these connections and will remove them from the list. Those connections are said to "time out."

```
debris:~> netstat -an | grep SYN | wc | awk '{print $1}'
0
debris:~>
```

Figure 5-10: Command to get a count of SYN packets received that have not been established into connections

5.2.4 UNIX Log Files

The two UNIX log files that ReaLLite examines are the sulog file and the utmp log file. The following paragraphs explain what data these log files contain, how ReaLLite accesses them, and what information is extracted for use in the IDS.

The SU Log File

This file is a record of all of the attempts of users to execute the "su" command. An entry is added to /var/adm/sulog each time su is invoked. The format of the entry is: "**SU date time result port user-newuser**." A result of "+" denotes a successful su, and a "−" denotes a failed attempt. The "port" refers to the name of the terminal that the user is connected to. The sulog file is only readable by root, but the inode data for this file can be read by any user. Only the inode data, not the contents of the sulog, is used by ReaLLite. Figure 5-11 shows a sample sulog file. It shows successful su's from user "kem" to root, from user root to kem, and from root to root. The last entry in the log file shows an unsuccessful su from user kem to root. While ReaLLite cannot read this file, it can issue a mid-level alert to the administrator that this file should be cross referenced with other data, if an illegal su is suspected. For example, if ReaLLite sees user kem start a root shell, and the sulog is modified, it will issue a mid-level alert that it appears that kem successfully su'd to root. After seeing this alert, the administrator can verify that there is an entry corresponding to a successful su to root by kem at that same time. The administrator can also verify that user the su to root was legal under the system's security policies.

```
SU 10/21 16:01 + pts/0 kem-root
SU 11/14 14:22 + pts/0 root-kem
SU 11/14 14:25 + pts/0 root-root
SU 02/11 14:49 + pts/0 kem-root
SU 02/26 11:49 + pts/0 kem-root
SU 03/08 15:09 + pts/0 kem-root
SU 04/25 00:45 - pts/0 kem-root
```

Figure 5-11: Sample sulog file

The UTMP Log File

The utmp log file holds accounting information about users currently logged into the system. It is located in /var/adm and is read by programs such as "who", "write", "w", and "login." ReaLLite uses information from this file to determine who is logged into the system. Figure 5-12 shows the data structure for this file. ReaLLite uses this data structure to read directly from the utmp binary log file. Among other information, it contains the login name, Process ID (PID) number, and device name of all users currently on the system. This data could be gathered with the "w" command, but that would involve spawning more processes and having to parse the output. Reading the data structure directly provides the same information with less overhead.

```
struct utmp {
     char  ut_user[8];      /* user login name */
     char  ut_id[4];        /* /sbin/inittab id (created by */
                            /* process that puts entry in utmp) */
     char  ut_line[12];     /* device name (console, lnxx) */
     short ut_pid;          /* process id */
     short ut_type;         /* type of entry */
     struct exit_status {
          short e_termination;  /* process termination status */
          short e_exit;         /* process exit status */
     }
     ut_exit;               /* exit status of a process
                            /* marked as DEAD_PROCESS */
     time_t ut_time;        /* time entry was made */
};
```

Figure 5-12: Data structure for entries in the utmp file

5.3 Multiple Scanning Levels

ReaLLite makes use of a low-level and a high-level scan in order to gather the needed inode information. Figure 5-13 shows a block diagram of the IDS, and how the different levels of scans are triggered. The DoS and probe detection is a separate cycle from the low-level and high-level scanning. One out of every three times the system wakes it goes through the DoS and probe detection cycle before heading to the scanning cycle. The scanning cycles consists of the low and high-level scans, which are used to detect illegal setuid root processes, buffer overflows that spawn root shells, and buffer overflows that modify a critical file on the system.

Dividing the scanning into two levels allows ReaLLite to avoid scanning in detail unless there is a reason to do so. Since ReaLLite was designed to be as lightweight as possible, it follows that scans should only be performed as needed. There is no reason to do the high-level scan of critical files for illegal modifications if the files cannot have been altered. As Figure 5-13 shows, the low-level scan determines if a high-level scan is needed. ReaLLite only scans critical files to detect buffer overflow attacks after a low-level scan reports that setuid root programs have been accessed. Likewise, ReaLLite does not perform the low-level scan unless there is reason to believe a setuid program was accessed. The following sections explain the two scan levels in more detail, as well as what triggers them.

5.3.1 The Low-Level Scan

The low-level scan is triggered by several events and results in a quick scan of the last access times of all setuid root programs. The three ways in which it is triggered are: a new connection, a new process, or a process that was missed.

1. **Triggered by new connection:** ReaLLite notices a new connection being established through data from netstat.

2. **New root process:** A new process is running that has a real or effective user ID of root. ReaLLite catches this new process in its scan of all active processes

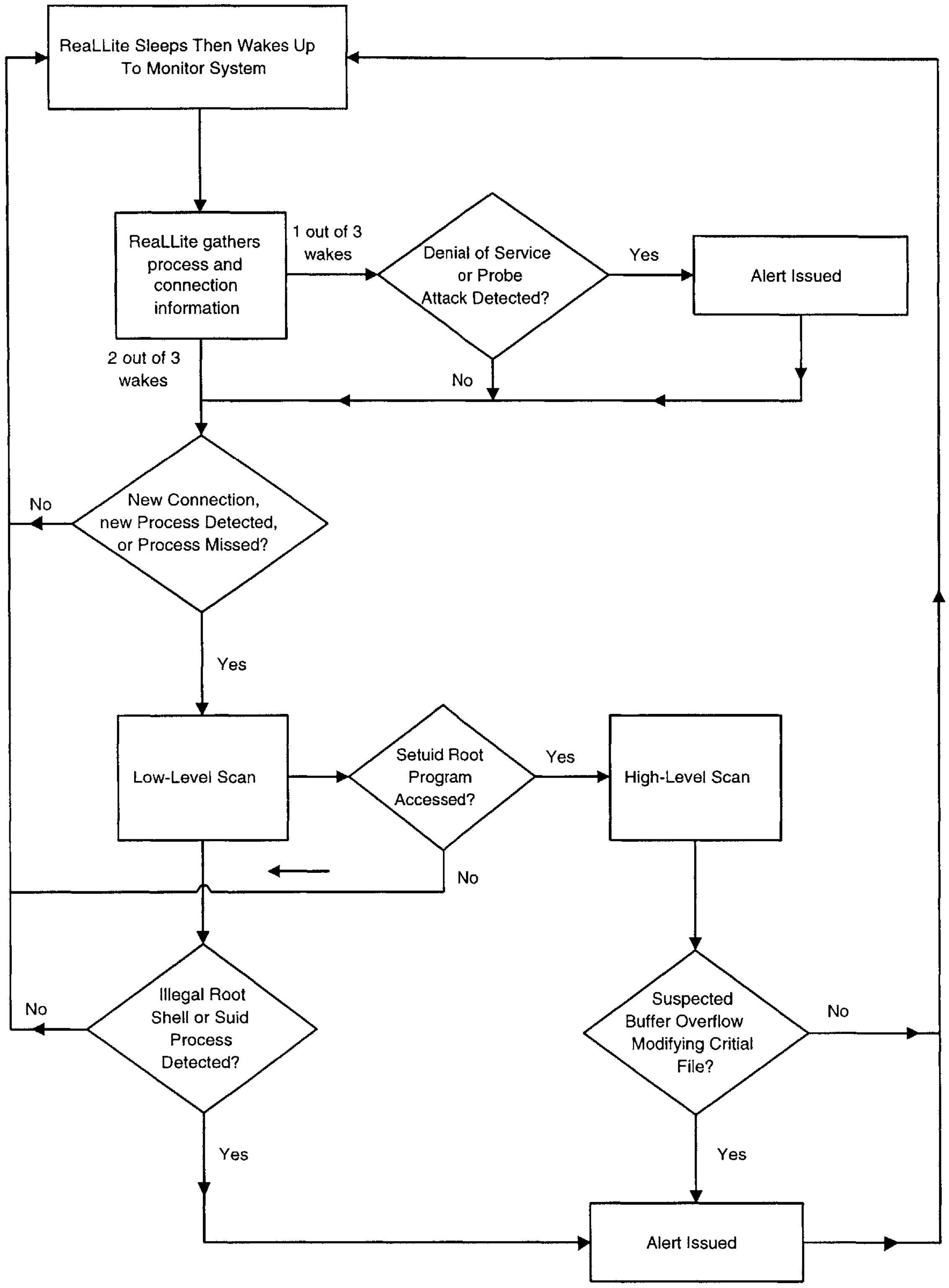
ReaLLite Sleeps Then Wakes Up To Monitor System
ReaLLite gathers process and connection information
1 out of 3 wakes
Denial of Service or Probe Attack Detected?
Yes
Alert Issued
No
2 out of 3 wakes
New Connection, new Process Detected, or Process Missed?
No
Yes
Low-Level Scan
Setuid Root Program Accessed?
Yes
High-Level Scan
No
Illegal Root Shell or Suid Process Detected?
No
Yes
Suspected Buffer Overflow Modifying Critial File?
No
Yes
Alert Issued

Figure 5-13: Block diagram of the ReaLLite system

with ps.

3. **Missed a process:** ReaLLite sees that the highest Process ID (PID) number is greater than it should be. This means that a process executed and died in between cycles of the IDS.

The low-level scan first verifies that there are no illegal root shell processes or unknown setuid root processes currently running. Through checking the inode data of the "su" and the sulog files, ReaLLite can verify that a root shell was created legally. Likewise, the low-level scan verifies that all new setuid processes running are in the list of setuid root files. If the new setuid process is not in the list, or if it is in the list but not accessed recently, ReaLLite issues an alert.

During the low-level scan, ReaLLite also examines the inode data of the files listed in the "suidlist" file. This list contains the full paths and filenames of all the setuid root programs on the system. The low-level scan checks to see if any of the setuid root programs were accessed since the last time ReaLLite checked. If any of the setuid root programs were recently accessed, the high-level scan is triggered.

5.3.2 The High-Level Scan

Triggered by the low-level scan, a high-level scan looks at critical files on the system to see when they were last modified, or had their inode data changed. This scan: (1) examines the modification times of all critical files, (2) correlates it with data from the low-level scan, and (3) compares the file and program with a list of known programs allowed to modify that file.

The first step of the high-level scan is to examine the modification time of the files in the "criticalfiles" list. The criticalfiles list contains a list of files that are considered critical to the security of the system. When the high-level scan is triggered, the inode data of all files in this list is examined to determine the last modify time and the time of the latest inode data modification. If any of these critical files have been recently modified or had their inode data altered, ReaLLite stores their name and what time the change took place.

The second step is to correlate this list of recently modified critical files with the last access times of setuid root programs, which was gathered in the low-level scan. If a setuid root program was accessed very shortly (i.e. within a second) before a critical system file was modified, then ReaLLite flags this event as a potential buffer overflow. ReaLLite makes note of the setuid program accessed and which critical file was modified directly afterwards.

The final step is to compare the suspicious activity to a list of executable programs that are legally allowed to modify critical files. Examples of these programs are "passwd", "login", and "su." This list was generated through a series of perl scripts when ReaLLite was first installed. The perl scripts take as input a list of critical files. The scripts then search through the man pages of system utilities looking for programs that modify those critical files. The scripts output a new file containing a list of all programs that can legitimately modify the critical files. The high-level scan uses this list before issuing an alert that a buffer overflow attack occurred in order to reduce the number of false alarms.

5.4 Order of Execution

ReaLLite is executed from the command prompt with no parameters. Simply running "./ReaLLite" is sufficient to start ReaLLite and send the output to standard out (stdout), which in this example would be the screen. The output may also be piped into a file. ReaLLite first looks for its configuration files: suidlist, criticalfiles, allowedfiles.lst. These files contain the list of all setuid root programs on the system, the files critical to the security of the system, and the programs allowed to modify those critical files. If these files are not present, the IDS tries to generate them as described earlier in this chapter.

After these files have been loaded (or created) ReaLLite uses ps and netstat to look at the current processes being run and the connections established. If ReaLLite sees a root process with a user owned parent process, the IDS checks information from the inode data of the su and sulog files to see those files were changed, implying that the user legally changed to root through the su command. ReaLLite also verifies that all setuid root programs that are running are legitimate setuid root programs (from the "suidlist" configuration file).

After ReaLLite verifies that the setuid root programs running are legitimate, as are any root shells, the IDS begins its cycle. It will sleep for a random number of seconds (between 0 and 10), then scan the processes and network connections again. ReaLLite also examines the /var/adm/utmp file by reading data from the utmp data structure directly. The utmp file contains a listing of who is logged on to the system currently. ReaLLite then looks at several things before determining whether to do a low-level scan of the system:

1. Whether a new network connection has been established, or is in the process of being established.

2. If there is a new process running with either a real or effective user ID of root.

3. If a process has been run and completed since the last low-level scan.

Any one of these events will cause ReaLLite to scan the last accessed time of all the setuid root programs on the system. If it determines that a program has been run since the last cycle of the system (or is running currently), then the high-level scan is called. This scan also looks for probes and Denial of Service attacks by looking at the detailed netstat data.

The high-level scan looks at the last modification and last inode status modification times of all of the critical system files. Lists are created of the modified files and the setuid programs run at the same time. This data is compared to a list of legal modifications to the critical files before ReaLLite determines whether a buffer overflow took place.

Chapter 6

Evaluating ReaLLite: False Alarms

Before the ReaLLite system was tested with real attacks, it was tested on many hosts to record background data. This data was gathered to tune ReaLLite to avoid issuing false alarms and to set appropriate thresholds for DoS and probe attacks. The hosts used for the tests were workstations and servers on a network at MIT Lincoln Laboratory. The network was located behind a firewall so there were presumably no attacks to taint the testing data.

The three main pieces of information for ReaLLite are process data from ps, network connection data from netstat, and inode data from the file system. Depending on whether the machine the IDS is running on is a workstation or a server, these pieces of information change quite frequently or not very often. A server is constantly handling new TCP/IP connections, which changes the netstat data and could start new processes to handle the connections. Workstations, on the other hand, have less constant activity and are very inactive when not in use.

6.1 Motivation for Testing

The two main motivating factors for testing the ReaLLite system were to reduce the number of false alarms and set the thresholds for different types of DoS and probe attacks. Because the machines running ReaLLite were behind a secured firewall, it was assumed that no attacks were being launched against the machines during the tests. Therefore, any alerts issued by ReaLLite were due to false alarms. False alarms encountered were alerting that an illegal root shell had been started or alerting that a buffer overflow had occurred resulting in a critical file being altered, when neither had actually taken place.

ReaLLite monitors the host's resources in order to try to detect attacks. During the testing phase, ReaLLite recorded statistics about system resources being used. These resources included: CPU utilization, memory used, total number of processes running, total number of connections established, number of TCP SYN packets received that had not been established into connections or timed out yet, and the number of TCP resets sent out.

6.2 The Test Bed

The test bed consisted of 10 workstations and two servers in a local area network at MIT Lincoln Laboratory. The machines were connected to the Internet through a firewall. All 12 machines were running Solaris 2.7 for the Sparc platform. ReaLLite ran on the host workstations for a one week period, and on the servers for several one week periods. Adjustments were made to the ReaLLite code between these periods to decrease false alarms.

6.3 Changes Made to ReaLLite for Testing

To test ReaLLite multiple times with the same data gathered, ReaLLite was modified to output the current data stored in memory regarding the host system on each cycle. This data was continuously appended to the end of a file to create a history of all input information. This information includes all process information gathered with ps, all network data gathered with netstat, information on recently accessed setuid root programs, and the data collected for DoS and probe attack detection.

A playback program was created that could read in the stored historical input data and play it back into the ReaLLite analyzer. This playback program runs the gathered data through the same rules for identification of attacks as data gathered in real-time. The playback program allowed the detection rules and thresholds to be modified, then re-tested with the same data.

6.4 False Alarms

Playing back the data gathered with ReaLLite allowed several causes of false alarms to be identified. Changes were made to the detection of illegal setuid root shells, buffer overflow detection, and DoS and probe attack thresholds based on data collected.

During the first week of testing, ReaLLite issued thousands of false alarms for buffer overflow attacks. When an intruder causes a buffer overflow, one of two things typically happens: a root shell is spawned, or a critical file has its contents or permissions altered. If a root shell is spawned, the detection is fairly straightforward and there is never a false alarm. If the intruder simply modifies the contents or permissions of a file, however, the detection is a little more subtle. ReaLLite detects these attacks by comparing the access times of setuid root programs to the modify and status change times in the inode data of the critical files. It was this latter type of attack that ReaLLite falsely reported had occurred.

Upon inspection of these alerts, it was determined that the cause was that the list of critical files contained files that were modified legitimately on a frequent basis. ReaLLite generated a list of critical files from the command: "find /etc > criticalfiles." This included files such as: /etc/.mnttab.lock, /etc/utmpx, and /etc/wtmpx. Similar files that ReaLLite false alarmed on were temporary files, log files, or symbolic links to log files that were constantly being modified by various processes. These files did not contain any critical system information, and their modification did not warrant an alert. The files were thus removed from the list of critical files to monitor and ReaLLite was tested on the workstations and servers again for another week.

During the second week, another source of false alarms was determined. The source of the false alarms was root shells legally spawned before ReaLLite started. When ReaLLite started on the server, it correctly identified the root shells, but was unable to verify that they were started legitimately. Because the users had su'd to root long before ReaLLite started, there was no way to check that the su file was accessed and the sulog file modified at the same time the shell started.

These false alarms were corrected by having ReaLLite only issue a warning when it starts if it sees root shells already running. ReaLLite alerts that the shells are present, and that it has no way of knowing whether they were started legitimately. It does not report them as buffer overflows, but does make note of them.

Another false alarm that was found a few times in the tests involved a setuid root program being legitimately accessed at the same time that a critical file is modified. This can occur if root is logged in and changing critical files legitimately at the same time a user is running a setuid program. Some setuid root programs, such as /bin/passwd modify a critical system file (/etc/passwd) when it is run successfully. a Similar false alarm could occur if a user is running setuid root programs legally while another user on the system is legally changing system critical files such as updating their password. ReaLLite tries to avoid these false alarms by taking into account when root is legally logged onto the system, and using a generated list of allowed modifications to critical files to cross-reference against suspected buffer overflow attempts. If a modification appears legal, no alert is issued. If the modification is not legal, but root is logged in legitimately, ReaLLite will issue only a mid-level warning. While an attack could occur while root is logged in, it is more likely that this is a false alarm caused by root making changes to critical files.

The testing period was also used to record normal usage patterns in order to establish reasonable thresholds for Denial of Service attacks. ReaLLite monitored CPU usage, the total number of processes and connections, the number of TCP reset packets sent out, and the number of SYN packets received but not timed out nor established into connections. Once the thresholds were established, they were used in the final week of testing, and in the testing of attacks, with no false alarms.

Figures 6-1, 6-2, and 6-3 show plots of CPU usage, processes, and connection values during testing. The x-axis of these graphs correspond to the number of low-level scans of the server. Since the low-level scans were only performed when ReaLLite detected activity on the server, rather than at set intervals, the x-axis reflects the data points gathered when the state of the system changed and not set units of time.

Figure 6-1 shows the highest CPU usage of any process over a period of over a week of testing ReaLLite on a server. The spikes in usage correspond to the nightly backups on the server. These spikes never go over 50% because the server was a dual processor machine, and a process can only use one CPU at a time. This means that the nightly spikes in CPU usage due to the backup utility were as high as possible. It was therefore not possible on this machine to detect a DoS attack that utilized all of the CPU time using a simple threshold, since the nightly backups did this legitimately.

Figures 6-2 and 6-3 show the total number of processes and connections for the server, respectively. The number of TCP connections established on the server stayed under 100, and the number of processes stayed under 160 for the vast majority of the time the system was monitored. If the number of processes or connections grows above the normal usage pattern, an alert should be issued about a possible resource exhaustion DoS attack. Based on the information gathered from these tests, a threshold of 200 processes and 120 connections was established for DoS detection.

The values for the SYN packets and TCP RST packets sent are dependent on time, they were measured at set intervals. This is because ReaLLite does not detect changes in these values in its low-level scan, so it must poll these values regularly. The data discussed in the following paragraphs was gathered over two days from a server.

Figure 6-4 show the number of SYN packets received that had not been established into TCP connections or timed out. This graph is taken at 10 second intervals, and the total number of SYN's received is never higher than 8. In a SYN Flood attack, the host will be hit with anywhere from 20 to thousands of connection attempts. A threshold of 12 SYN's received, but not established into connections, was set as the DoS threshold from the data collected. Mutaf's paper [16] performs a similar analysis on the number of SYN's received per second by a server. His values of 15-20 SYN's per second as a threshold for a server are slightly higher, but include SYN's that were sent legitimately and established into connections, which ReaLLite does not include.

Figure 6-5 contains the number of TCP RST packets the host sent during 10 second intervals. This figures covers a total time period of about two days. The reset packets were sent out at an average rate of around one TCP RST per minute. In a port scan, thousands of ports might be scanned in just a few minutes, orders of magnitude above the average rate. This leaves room to set a low threshold and catch slightly slower port scans. The threshold used to detect a port scan during the testing of ReaLLite was 50 TCP RST's sent in a thirty second period. If more than 50 TCP RST's were sent within thirsty seconds, ReaLLite issued an alert that the host was being port scanned. The maximum TCP RST's in an interval was about 25. This occurred near the 12 hour and 36 hour marks, corresponding to around 4 a.m. This is consistent with a nightly system maintanence job, and was considered normal network traffic.

The values mentioned here were used in both the last stage of the testing for false alarms, and in the actual attack testing. No false alarms resulted with these thresholds.

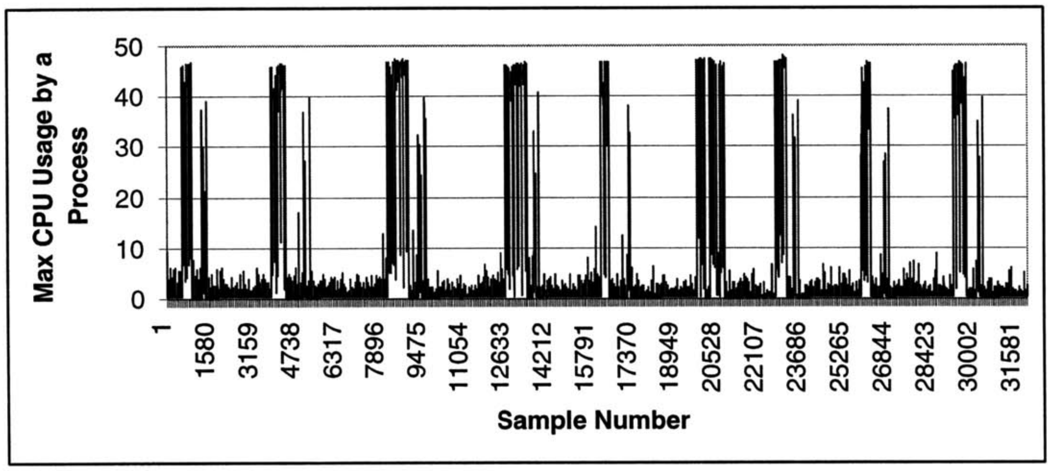

Figure 6-1: CPU usage for a server over a week

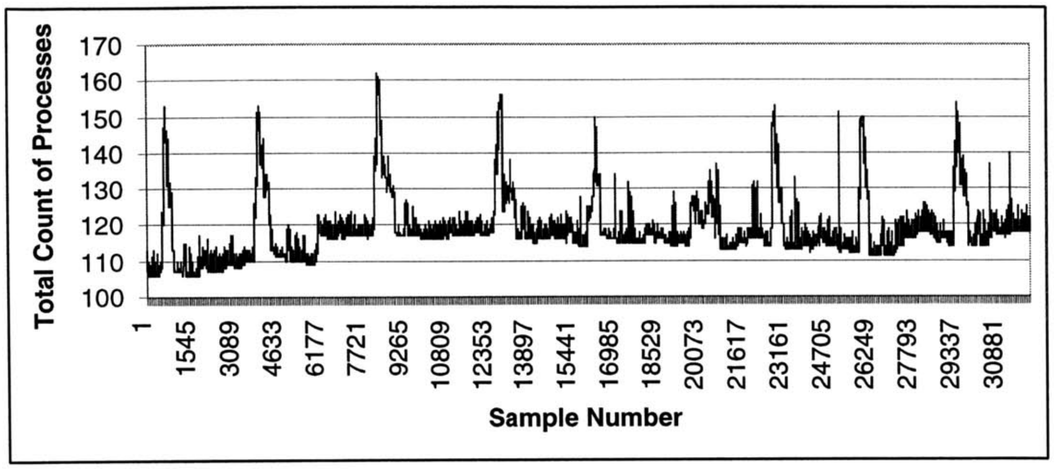

Figure 6-2: Number of processes running on a server

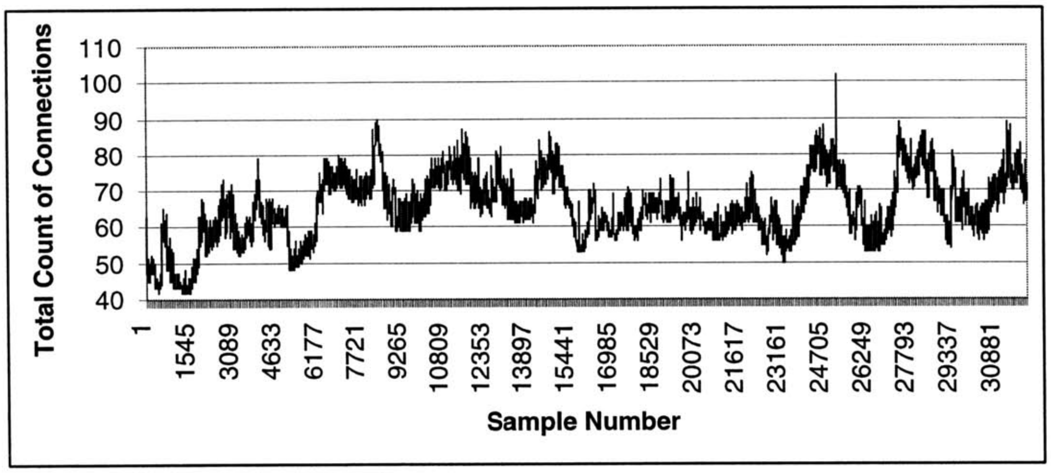

Figure 6-3: Number of connections established

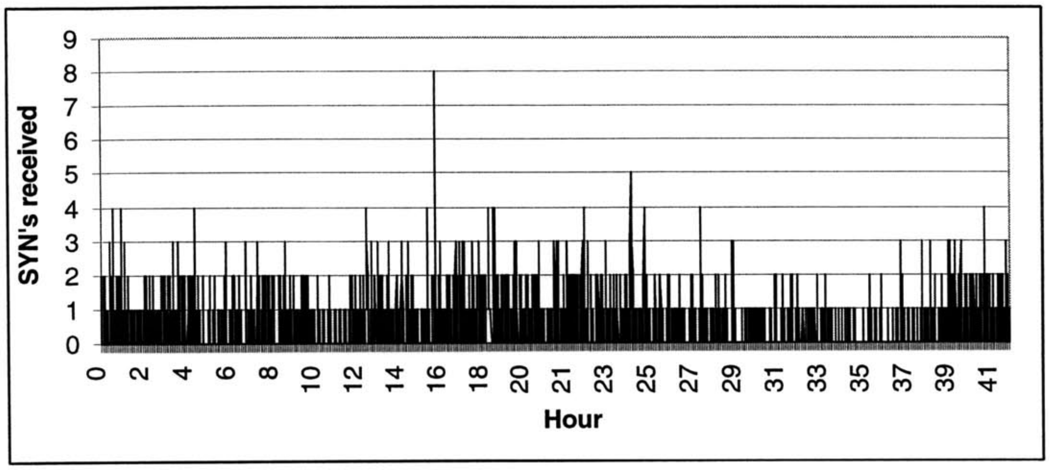

Figure 6-4: TCP SYN packets received that had not timed out or been established as connections

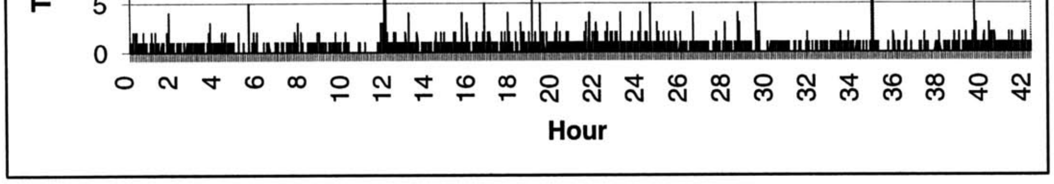

Figure 6-5: Number of TCP resets packets sent out in 10 second intervals

6.5 Summary of Modifications

The following list summarizes the modifications made during training to reduce false alarms and insure good detection. After making these changes to ReaLLite, there were no false alarms in one week of testing.

1. Eliminate files from the critical files list that are not critical, or that get modified constantly. This includes temporary lock files and log files in /etc. These files were the primary cause of false alarms, and never the target of an attack.

2. Reduce the time difference between setuid access and critical file modification. Through testing buffer overflow attacks that modified files it was determined that the overflow is almost instantaneously after the execution of the setuid root program. This causes the file modification time to be at virtually the same time as the setuid root program access time. By flagging only those file modifications that occur within a second of a setuid root program access time we are able to reduce the number of false alarms to zero.

3. Generate list of setuid root programs allowed to modify critical files. Perl scripts were used to parse through man pages and find the files that are legally modified by system programs. When ReaLLite notices that a a critical file has been modified directly after a setuid root program has been accessed, this list of "allowed modifications" is cross-referenced. If the modification appears valid, the IDS does not report a buffer overflow has occurred, but does make a note of the incident in the log file.

4. Take into account who is logged into the system. In order to reduce the false alarms called by the coincidental execution of a setuid root program at the same time that root is legitimately changing critical system files, ReaLLite reads the utmp log file to determine who is currently logged into the system and whether they are active or idle. If root is logged in and active then a warning is issued, but ReaLLite reports that it is more likely that root was editing system files than that an intruder attempted a buffer overflow.

5. Run private versions of ps and netstat. Since these programs are used by ReaLLite, they are constantly having their time stamps updated. This makes detecting an attack that performs a buffer overflow on them very difficult. By having private versions of these files for ReaLLite, the system can use the private copies and not change the inode data for the system-wide copies. This was the last change made to the ReaLLite system, and was motivated by the use of a ps exploit in the 1999 DARPA Intrusion Detection Evaluation. Since ReaLLite uses ps in each cycle, the last access time in the inode data of ps is constantly getting updated to the current time. This makes detecting another user running the ps program very difficult. By providing ReaLLite with its own copy of ps, the system is able to differentiate between its own use of ps and another use executing the program.

6.6 Summary

The testing of ReaLLite on workstations and servers provided invaluable data for eliminating false alarms and setting accurate thresholds for Denial of Service and probe attacks. By eliminating unnecessary and constantly changing files from the critical files list, and cross-referencing suspected buffer overflows with a list of allowed modifications to critical files, false alarms were eliminated. The thresholds established for the DoS and probe attacks also did not yield any false alarms in the final week of testing.

Chapter 7

Evaluating ReaLLite: Attack Detection

As noted in Chapter 6, some types of background traffic result in false alarms from ReaLLite. The system was modified to eliminate all false alarms on all of the recorded data, and then tested against real attacks. A total of 12 attacks were used against ReaLLite. Seven of these attacks were from the 1999 DARPA Intrusion Detection Evaluation [10], and five were new attacks. ReaLLite detected all of these attacks except one, which crashed the host before ReaLLite could issue an alarm. While this detection is very good, ReaLLite could still be improved. A attacker who knows how ReaLLite gathers and uses data could perform a stealthy attack that ReaLLite would miss.

7.1 Attacks Tested Against ReaLLite

ReaLLite was tested with attacks that altered some data that ReaLLite monitored, so that ReaLLite had a chance of catching them. The attacks used were specific to the Solaris Operating System, for either the Sparc or x86 version. These attacks performed actions that a host-based IDS should detect, such as starting illegal root shells, modifying critical system files, and performing Denial of Service attacks on the host. Seven of the 12 attacks came from the 1999 DARPA Intrusion Detection

Evaluation. The other five attacks were new exploits found on the BugTraq mailing list archive [2] and the attack database on the SecurityFocus web site [21]. Tables 7.1 lists the old attacks used, and 7.2 contains the new attacks. Both tables include the action performed by the attack and the evidence left behind.

Vulnerability	Program	Action	Evidence
buffer overflow	eject	Creates a root shell	New root shell created by user
buffer overflow	ffbconfig	Creates a root shell	New root shell created by user
buffer overflow	fdformat	Creates a root shell	New root shell created by user
buffer overflow	ps	An unreadable file from an unreadable directory is copied	Unreadable file is accessed
denial of service	neptune	Victim is flooded with SYN packets	Abnormally high number of SYN packets received
denial of service	process table	Fills up process table of victim	Extremely high number of processes running
probe	nmap	Ports are scanned to determine if open	Large number of ports are scanned

Table 7.1: Attacks tested against ReaLLite from the 1999 DARPA Intrusion Detection Evaluation

7.2 Attacks From the 1999 Intrusion Detection Evaluation

The following attacks were used against Solaris systems in the 1999 Intrusion Detection Evaluation at MIT Lincoln Laboratory. They include U-2-R buffer overflow attacks, DoS attacks, and probe attacks.

7.2.1 eject

Description: The setuid root program /usr/bin/eject distributed with Solaris 2.5 is susceptible to a buffer overflow attack. The eject program unmounts and ejects removable media from devices that do not have an eject button. These devices are managed by Volume Management. The volume management library, libvolmgt.so.1 performs insufficient bounds checking, allowing the internal stack space of the eject program to be overwritten. This vulnerability permits the execution of arbitrary code with super user privileges. The specific exploit used in the tests spawned a root shell after overflowing the eject binary [10].

Attack Signature: This attack resulted in a shell process being created with real and effective user ID's of root. The owner of the parent process was not root, the su file was not accessed, and sulog was not modified.

7.2.2 ffbconfig

Description: The setuid root program /usr/sbin/ffbconfig distributed with Solaris 2.5 is susceptible to a buffer overflow attack. The ffbconfig program configures the Creator Fast Frame Buffer (FFB) Graphics Accelerator, which is used when the FFB Graphics accelerator card is installed. Insufficient bounds checking on the arguments passed to the program allows the stack space to be overwritten. Arbitrary code with super user privileges can be run. Much like the eject exploit, the specific code used in the tests spawned a root shell [10].

Attack Signature: This attack resulted in a shell process being created with real and effective user ID's of root. The owner of the parent process was not root, the su file was not accessed, and sulog was not modified.

7.2.3 fdformat

Description: The setuid root program /usr/bin/fdformat distributed with Solaris 2.5 is susceptible to a buffer overflow attack. The fdformat formats diskettes and PCMCIA memory cards, and also uses the volume management library, libvolmgt.so.1. Just like the eject vulnerability, the exploit code used here overwrites the internal stack space of the fdformat program, spawning a root shell [10].

Attack Signature: This attack resulted in a shell process being created with real and effective user ID's of root. The owner of the parent process was not root, the su file was not accessed, and sulog was not modified.

7.2.4 SYN Flood (Neptune)

Description: This is a SYN Flood Denial of Service attack. All TCP/IP implementations are vulnerable to some degree to a SYN Flood attack. In establishing a TCP/IP connection, the server must set aside some finite amount of memory for storing the connection information in a data structure of pending connections. An attacker can attempt to fill this data structure by flooding the server with "half-open" connection attempts (connection attempts that never fully establish a connection). The server will usually remove "half-open" connection attempts from the data structure after a set time limit, but if the attacker can fill the structure faster than the attempts expire, the server will be unable to accept new connections. The server may crash from this attack, exhaust memory, or otherwise become inoperative [10].
Attack Signature: The victim machine receives a large number of half-open TCP connections. These show up as SYN packets received, and answered, but without an ACK received. The exact number of SYN packets received may vary. ReaLLite was tested multiple times with 20 to 250 SYN packets sent within a few minutes.

7.2.5 ps

Description: The setuid root program ps distributed with Solaris 2.5 is susceptible to a buffer overflow attack. The ps program is used to list information about active processes on a system. Ps performs improper bounds checking, allowing an attack to execute arbitrary code with super user privileges. In the specific exploit used for the tests, a secret file was copied from a directory that the user did not have access to read [10].
Attack Signature: The permissions on a non-readable file and non-readable directory are changed to be world-readable. A secret file is copied into the home directory of the user executing the attack, and the permissions on the file and directory are

changed back. No other changes are made to the system.

7.2.6 nmap

Description: Nmap is a probing tool used to perform a port scan of a machine or even an entire network. Nmap supports a variety of flavors of port scans including a TCP connect() scan, SYN, FIN, Xmas Tree, Null, and Ping scanning. Nmap also allows scanning with UDP, RPC, or even an FTP bounce attack. Scans are highly configurable, allowing the user to specify which ports to scan, whether to scan ports sequentially or randomly, and how long to wait in between scanning each port. Nmap also has the ability to fragment IP packets, spoof scans from other IP addresses to hide the real scan, and perform operating system identification via TCP/IP fingerprinting [10].

Attack Signature: Nmap tries to connect to a number of ports. If these ports are open, a half-open or full connection is made. If these ports are close, a RST packet is sent out. Depending on the speed of the scan, thousands of ports may be scanned in a minute. During the probe attack on the ReaLLite system, 1078 ports were scanned in under 30 seconds.

7.2.7 process table

Description: The process table attack was a novel DoS attack developed specifically for the 1999 DARPA Intrusion Detection Evaluation. The attack consists of establishing many TCP/IP connections to a server. These connections are usually handled by setuid root daemons, which then fork() a process. A problem arises, however, when the server forks() enough processes as to fill its process table. Once the process table is filled, no more processes can be started on the server [10].

Attack Signature: An usually large number of processes start running.

7.3 New Attacks

New attacks were gathered from web sites and the BugTraq mailing list in order to further test the ReaLLite system. The new attacks used include a R-2-L buffer overflow attack, U-2-R buffer overflow attacks, and a DoS attack. The following paragraphs describe these attacks used.

Vulnerability	Program	Action	Evidence
buffer overflow	netpr	Changes a shell file to setuid root	New setuid root file on the system
buffer overflow	xsun	Creates a root shell	New root shell created by user
buffer overflow	sadmind	Allows execution of arbitrary code	Varies depending on command executed
buffer overflow	lpset	Creates a root shell	New root shell created by user
denial of service	fork bomb	Fills up process table of victim	Extremely high number of processes running

Table 7.2: New attacks tested against ReaLLite

7.3.1 netpr

Description: The netpr program distributed with Solaris 2.6 and 2.7 is susceptible to a buffer overflow attack. Netpr, part of the SUNWpcu package, prints a print job by opening a network connection to the printer and sending the data to the printer via the BSD print protocol or a TCP pass-through. Due to insufficient bounds checking for the argument of the printer name, the internal stack space can be overwritten and arbitrary code executed. The exploit code used in the tests changed an executable shell located in the temporary directory /tmp to be setuid root. When this shell was then executed, a user would have control of a root shell.

Attack Signature: This attack changes the permissions on an executable shell in the temporary directory. When the shell is executed, the user is seen executing a setuid root program.

7.3.2 xsun

Description: The setuid root program /usr/openwin/bin/Xsun distributed with Solaris 2.6, 2.7, and 2.8 is susceptible to a buffer overflow attack. Xsun is the standard X11 server shipped with Solaris. Due to insufficient bounds checking on the argument used to select the output device, the internal stack space is overwritten and arbitrary code can be executed. The exploit code used in the tests spawned a root shell.
Attack Signature: This attack resulted in a shell process being created with real and effective user ID's of root. The owner of the parent process was not root, the su file was not accessed, and sulog was not modified.

7.3.3 sadmind

Description: The setuid root program /usr/sbin/sadmind distributed with Solaris 2.6 and 2.7 is susceptible to a buffer overflow attack. Sadmind is the daemon used by Solstice AdminSuite applications to perform distributed system administration operations. Part of the /usr/snadm/lib/libmagt.so.2 library performs insufficient bounds checking. When executed with properly chosen arguments, the stack pointer is overwritten and arbitrary code may be executed. The exploit code used in the tests allowed the user to supply the exact command to be executed with root privileges. This attack was ranked the fifth most severe of all computer attacks by the CERT® Coordination Center [3].
Signature: The signature of this attack depends on the actual code chosen by the attacker to be executed. The exploit allows a remote user to issue any command as root on the local system. The exact code used in the evaluation of ReaLLite added a new entry to the /etc/passwd and /etc/shadow files.

7.3.4 lpset

Description: The setuid root program /usr/bin/lpset distributed with Solaris 2.6 and 2.7 is susceptible to a buffer overflow attack. Lpset is a standard utility for setting the printing configuration information in the system configuration database. Due to insufficient bounds checking on an argument, the internal stack space of lpset can be overwritten and arbitrary code executed with root privileges. The exploit code used in the tests spawned a root shell after overflowing lpset. Figure 7-1 shows the execution of the lpset exploit code used in testing the ReaLLite system. The exploit performs a buffer overflow to elevate the user "kem" to root.

Attack Signature: This attack resulted in a shell process being created with real and effective user ID's of root. The owner of the parent process was not root, the su file was not accessed, and sulog was not modified.

```
Last login: Sun May 13 14:43:23 2001 from budweiser.mit.ed
Sun Microsystems Inc.   SunOS 5.6       Generic August 1997
No mail.
Sun Microsystems Inc.   SunOS 5.6       Generic August 1997
debris:~> id
uid=10001(kem) gid=10(staff)
debris:~> ./lpsetex
ret: 0xefffad90 xlen: 17998 ofs: 0x4e90 (20112)
# id
uid=0(root) gid=10(staff)
#
```

Figure 7-1: Root shell spawned from an lpset exploit

7.3.5 fork bomb

Description: This is a local DoS attack that consists of a simple fork() bomb. The attack code is a script that calls itself twice, forking both new processes into the background. These two new processes call themselves again, and the process continues until no more processes can be started. Figure 7-2 shows how the attack script was generated on the victim machine.

Attack Signature: Processes are created until the maximum number was reached. CPU usage also approached 100%.

```
dropout:~> cat > forkbomb
#!/bin/sh
$0 & $0 &
dropout:~> chmod +x forkbomb
dropout:~> ls -l forkbomb
-rwxr-xr-x    1 kem      kem            20 May 14 00:24 forkbomb*
dropout:~>
```

Figure 7-2: A fork bomb resource exhaustion attack

7.4 Detection Results For the 1999 Intrusion Detection Evaluation Attacks

All of the attacks from the 1999 Intrusion Detection Evaluation were detected successfully. Table 7.3 lists the results of the attacks on a system running ReaLLite, and how the attacks were detected. The eject, ffbconfig, and fdformat exploits all spawned root shell processes after overflowing their respective setuid root programs. ReaLLite detected the buffer overflows right after they occurred since a process with real and effective user ID's of "root" is started with a parent owner other than root. With no modification to the sulog file, this cannot be a legal su to root. The event is flagged at the highest warning level, with ReaLLite reporting the time, the setuid root program overflowed, and the user who did it.

The neptune SYN Flood exploit was also detected. ReaLLite was able to determine in less than a minute that the number of SYN_RCVD packets of the system was unusually high, indicative of a SYN flood attack. Figure 7-3 shows the attack being launched from a remote machine and being detected by ReaLLite on the victim machine. Figure 7-4 shows SYN packets received but not established into full connections or timed out yet. The attack is clearly visible. This attack will be detected as long as there are 12 or more SYN packets received that have not timed out or been fully established into connections.

The ps attack used a buffer overflow to gain privileges and copy a secret file out of a restricted directory. ReaLLite was able to detect the modification of the permissions on the directory containing the secret file. This directory was listed as one

```
dropout:~# ./neptune

USAGE: ./neptune
         -s unreachable_host
         -t target_host
         -p port [0 all ports]
         -a amount_of_SYNs
dropout:~# ./neptune -s 10.0.0.1 -t debris -p 22 -a 250

Flooding target:                2063884818
On port:                        22
Amount:                         250
Puportedly from:                16777226
......................................................................
......................................................................
......................................................................
........................................
dropout:~#
```

```
debris:~> ./reaLLite
989811790 [0] reaLLite ids starting up now
989811792 [1] new connection (or attempt) detected
989811792 [1] missed a process
989811792 [1] first layer scan triggered
989811792 [1] first layer scan completed
989811865 [9] SYN FLOOD DETECTED WITH 250 SYN_RCVD's
```

Figure 7-3: ReaLLite successfully detecting a neptune SYN flood

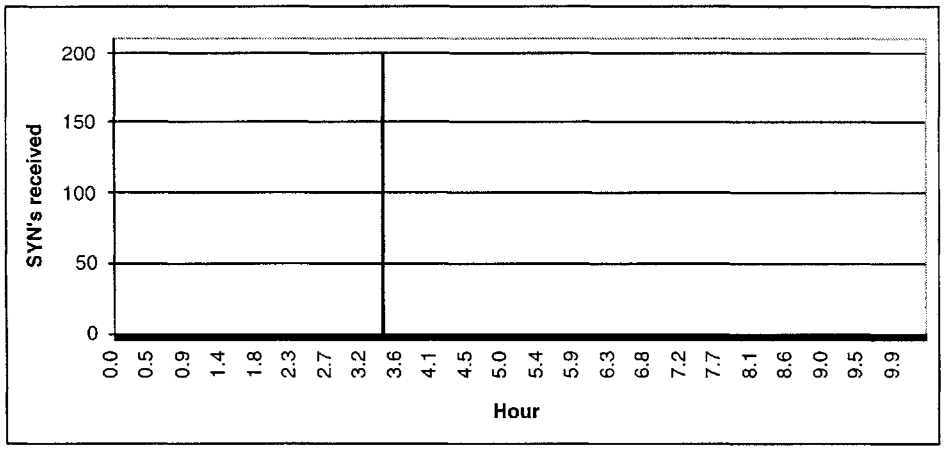

Figure 7-4: Graph of SYN packets received but not timed out or established into connections yet

Attack	Platform Tested	Detected by ReaLLite	How Attack Was Detected or Why it Went Undetected
eject	Solaris 2.5 SPARC	Yes	Illegal root shell detected
ffbconfig	Solaris 2.5 SPARC	Yes	Illegal root shell detected
fdformat	Solaris 2.5 SPARC	Yes	Illegal root shell detected
ps	Solaris 2.5 SPARC	Yes	Copying of secret file not detected
nmap	Solaris 2.6 SPARC	Yes	Large number of outbound TCP resets detected
process table	Solaris 2.6 SPARC	Yes	Large number of processes detected
neptune	Solaris 2.6 SPARC	Yes	Large number of TCP connection attempts detected

Table 7.3: Results of 1999 attacks on a host running the ReaLLite IDS

of the critical files that ReaLLite monitors, since access to files in that directory was restricted. ReaLLite was able to catch the inode data status change of the directory and report the buffer overflow attack. This result demonstrates that detecting modifications of secret files is possible as long as the directory containing the files is accessable to other users.

Detection of an nmap scan relies on the fact that very few TCP RST packets are sent from an end-host unless it is being probed by a port scanner. In the process of being scanned, the end-system sends RST packets whenever the scanner attempts to connect to ports that are not open. In the tests, nmap was called with a variety of options, including scanning only the first 1024 ports, and only those matching standard protocols. ReaLLite consistently detected that a port scan was taking place. Figure 7-5 shows an nmap scan being launched against a remote system, and ReaLLite detecting the host being port scanned. Figure 7-6 shows the graph of TCP Reset packets sent out over time, with the attack clearly visible. Scans will be detected as long as there are 50 resets sent per minute. If the nmap probe were slowed down to under this threshold, such as 10 ports scanned per minute, then ReaLLite would not detect it.

To be successful, the process table exploit makes connections to services controlled

```
dropout:~# nmap -sS -F -O debris

Starting nmap V. 2.3BETA5 by Fyodor (fyodor@dhp.com, www.insecure.org/nmap/)
Interesting ports on debris.mit.edu (18.98.4.123):
Port     State       Protocol  Service
22       open        tcp       ssh
25       open        tcp       smtp
111      open        tcp       sunrpc
4045     open        tcp       lockd
6000     open        tcp       X11

TCP Sequence Prediction: Class=random positive increments
                         Difficulty=44626 (Worthy challenge)
Remote operating system guess: Solaris 2.6 - 2.7

Nmap run completed -- 1 IP address (1 host up) scanned in 5 seconds
dropout:~#
```

```
debris:~> ./reaLLite
989813341 [0] reaLLite ids starting up now
989813344 [1] new connection (or attempt) detected
989813344 [1] missed a process
989813344 [1] first layer scan triggered
989813344 [1] first layer scan completed
989813360 [9] PORTSCAN DETECTED WITH 1017 NEW RESETS
```

Figure 7-5: ReaLLite successfully detecting an nmap port scan

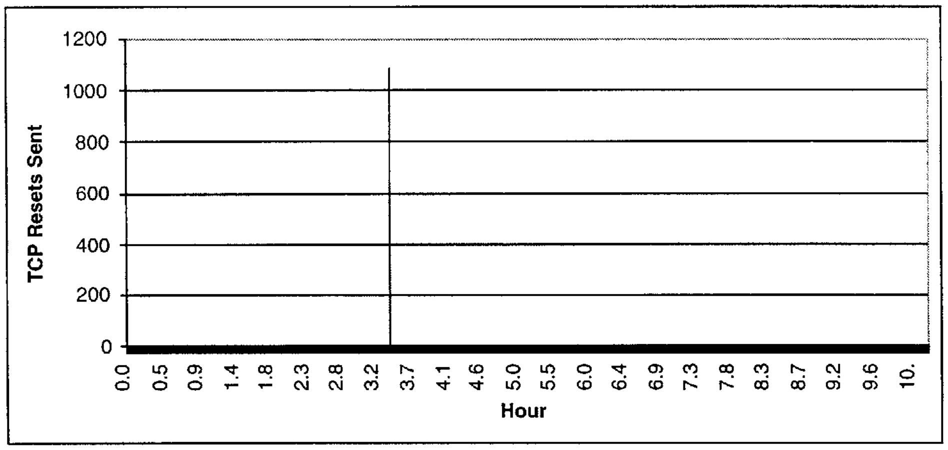

Figure 7-6: TCP reset packets sent out by the host over time

by inetd. The connections must not exceed a certain number per minute. By spacing out the connection enough to slip by inetd, this attack ended up taking several minutes to complete. ReaLLite was able to detect the attack while it was in progress and before the process table had been filled. The threshold for the total number of processes was set at 200, and it took this attack around 10 minutes to reach that threshold, with a four second delay in between connections.

7.5 Detection Results For New Attacks

Table 7.4 shows the results of the new attacks tested against the ReaLLite system. ReaLLite successfully detected all the attacks except for one, which froze the host. The following paragraphs further discuss these attacks.

Attack	Platform Tested	Detected by ReaLLite	How Attack Was Detected or Why it Went Undetected
netpr	Solaris 2.6 x86	Yes	Illegal setuid root process detected
xsun	Solaris 2.6 x86	Yes	Illegal root shell detected
sadmind	Solaris 2.6 x86	Yes	Illegal critical file modification detected
lpset	Solaris 2.6 x86	Yes	Illegal root shell detected
fork bomb	Solaris 2.6 x86	No	Host froze before ReaLLite could issue a warning

Table 7.4: Results of new attacks on a host running the ReaLLite IDS

The lpset and xsun attacks spawned root shell processes and were similar to the eject, fdformat, and ffbconfig attacks from the 1999 Intrusion Detection Evaluation. ReaLLite detected both buffer overflows right after they occurred. In each attack a process with real and effective user ID's of root was started, and the parent of the process was not root. There was no modification to the sulog file, and su was not accessed. The event was therefore reported with the highest warning level. ReaLLite also reported the time of the attack, the setuid root program overflowed, and the user who performed the attack.

Figure 7-7 shows the ReaLLite system successfully detecting the lpset exploit used in the test. The IDS first notices a new root process, then detects that it was spawned by a user without the use of su.

```
debris:~> ./reaLLite
989809190 [0] reaLLite ids starting up now
989809193 [1] new connection (or attempt) detected
989809193 [1] missed a process
989809193 [1] first layer scan triggered
989809193 [1] first layer scan completed
989809221 [1] new root process, pid 29152
989809223 [1] missed a process
989809223 [1] first layer scan triggered
989809223 [3] parent process not owned by root for 'sh' pid:29152 (& uid/ruid = root)
989809223 [3] user 'kem' spawned the new program with uid/ruid = root
989809223 [9] su not accessed within threshold - overflow likely
989809223 [9] sulog not modified within threshold - overflow likely
989809223 [1] first layer scan completed
```

Figure 7-7: ReaLLite successfully detecting an lpset root exploit

Instead of immediately spawning a root shell, the netpr exploit modifies the permissions on a shell file (the ksh shell) placed in the temporary directory /tmp. The permissions are changed on /tmp/ksh so that it is a setuid root program. A root shell is provided when this shell is executed by a user. This attack was not detected when the netpr exploit was initially run since no critical system files were changed, however, as soon as the user executes the setuid root shell, ReaLLite detects it. ReaLLite has a list of the setuid root programs on the system, so when it detects a new one running it sets off an alarm.

The sadmind exploit allows for the user to specify what code to run with root privileges. For the purposes of testing, the code run as root consisted of adding entries to the /etc/passwd and /etc/shadow files. ReaLLite detected that sadmind was accessed and the critical files passwd and shadow were modified.

7.5.1 Attacks Not Detected

The only attack not detected was the fork bomb attack from the new set of attacks. This attack halted the system in a matter of seconds. ReaLLite did not have a chance to examine the process data before the machine froze. The machine was rendered unusable at that point, and had to be rebooted to be restored. If the attack had been slower, the IDS would have been able to detect the unusually high process count and report it along with the user who executed the code, although ReaLLite would be unable to stop the attack.

7.6 Improvements Needed

Analysis of attacks and detection rules suggests several approaches to improve ReaLLite. Possible improvements include analyzing more types of network information, tracking user's actions, and notifying administrators when attacks occur. These improvements are discussed in the following paragraphs.

In this version of ReaLLite, network data was used to trigger the scanning of setuid root programs on the system. Little of the data provided by netstat, however, was used. Netstat will report on the ports that are open and listening for connections, for instance. ReaLLite could incorporate this information and check for backdoors listening for a connection.

Netstat also provides low level data about TCP, UDP, IP, and ICMP packets. Using some of this information could help ReaLLite detect problems with the network itself, such as a large number of checksum errors in packets, packets expiring because of short TTL's, and a large number of fragmented packets arriving out of order. Some of these problems may indicate attacks that use evasion and insertion of data methods against network-based IDS's.

ReaLLite successfully determines which user performed a buffer overflow if it was a local attack, and if a critical file was changed with that overflow. A useful extension would be to track what a user does, especially if the buffer overflow launches a root shell. Since ReaLLite is not running as root and cannot actively stop an attack, it would help if ReaLLite could track everything a user does after overflowing a setuid root program so that the administrator can get a better understanding of the effect of the attack. Through process information, and the use of the "w" command, ReaLLite could monitor activities of a user and store this possibly incomplete record.

Finally, since ReaLLite is a real-time system it can detect attacks as they are happening. It therefore makes sense that ReaLLite should contact an administrator as an attack is taking place. This functionality could be added quite easily. ReaLLite could send e-mail to any number of people, display an alert on the console, or use the UNIX syslog system to send a message when it notices an attack, along with detailed information about the attack.

7.7 How Stealthy Attacks Could Evade ReaLLite

With knowledge of how ReaLLite gathers and analyzes data, stealthy attacks could avoid detection. Some of these stealthy methods include accessing critical files instead of modifying them after a buffer overflow, creating a trojan setuid root program, sleeping after a buffer overflow, faking a legitimate "su," and performing a very slow port scan. The following paragraphs discuss how these stealthy attacks could evade detection by the ReaLLite system.

One difficult operation for a system like ReaLLite to monitor is the copying of files. When a file is copied, its contents are only read, not modified. Many programs legitimately read the contents of critical files on the system. When you log in, for instance, /etc/passwd and /etc/shadow are read. A clever intruder can therefore craft a buffer overflow that simply copies the contents of a file such as the /etc/shadow file. ReaLLite only sees that a critical file was read, which could have been caused by any number of things. If intruders have a copy of the shadow file, they cannot necessarily log in as another user, but they might be able to recover some of the passwords by using a brute force attack against the file.

The netpr attack used in the tests changed the permissions on a shell in the /tmp directory to create a setuid root shell when executed. ReaLLite caught this attack when the new setuid root program was executed. If the exploit had copied this setuid root shell from /tmp/ksh to the location of a legitimate setuid root program, such as /usr/sbin/ping, then ReaLLite would not have flagged the attack. This is because it would have appeared to ReaLLite that the user was simply running ping legitimately when in fact they would have been accessing the root shell created by the overflow attack.

If an overflow modifies a file, ReaLLite detects this by looking at the last modification and status change times of the critical files in comparison to the access times of setuid root programs. If an intruder knows this, the exploit code can be made to sleep for a few seconds before altering a critical file. This results in the stamps of the last access time of the setuid root program and the last modified time of the critical file to be different and allow the attack to go undetected.

ReaLLite determines that an active root shell was started illegally if there was no modification to sulog at the time of the start of the shell and /usr/bin/su was not accessed. The log modification would correspond to an entry saying that the su was legal. If an intruder spawns a root shell, then very quickly runs "su" as root, it will update the access time of /usr/bin/su as well as add an entry to the sulog stating that root su'd to root. This essentially "spoofs" a legal su and fools ReaLLite if executed quickly enough.

A port scan is detected based on the number of TCP reset packets that the host sends out. These reset packets do not occur often except in the case where the machine is being port scanned. An attacker may perform a stealthy scan that has long delays between connection attempts. This scan might take hours, days, or even weeks to complete. Such a scan would fall well below the threshold that ReaLLite uses to determine how many TCP RST's are enough to constitute a port scan.

7.8 Warning Levels

ReaLLite issues a warning when it detects behavior that is suspicious. This warning contains the time of the incident, what ReaLLite thought happened, and an associated warning level. There are currently three warning levels: low-risk, mid-risk, and high-risk.

1. **Low-risk:** The events logged at this level are not real security risks. They consist of the ReaLLite system starting up or shutting down, new processes

or connections being established, or files not being found. If the ps program cannot be found, for instance, ReaLLite will exit with a low-risk warning level, explaining that the ps file was not found.

2. **Mid-risk:** While these events by themselves should not be considered a security risk, they might be suspect or part of an attack. These events include a new root process being spawned, even legitimately in an su, a new root process being spawned without a living parent process (parent process died before ReaLLite could identify it), and an allowed modification to a critical system file.

3. **High-risk:** These events consist of alerts of attacks. They include: buffer overflow attack, unknown setuid root program running, a port scan probe, a SYN flood, or a resource exhaustion Denial of Service attack.

Chapter 8

ReaLLite Ported to Linux

ReaLLite was developed on the Solaris Operating System. As a proof-of-concept of ReaLLite's portability, the system was ported to the Linux Operating System. Linux is a UNIX system that is run primarily on Intel-based PC's, although it also runs on a number of other platforms, including Sparc, Alpha, and Macintosh. ReaLLite had to be modified slightly to run on Linux instead of Solaris, but the changes were minor. The ReaLLite Linux system was tested with SYN Flood and port scan attacks, both of which were detected.

8.1 Changes Made

Several small changes had to be made to the ReaLLite system. These changes involved the arguments passed to the ps and netstat programs, the location of log files, the location of setuid root programs, and critical files of the system. The ps program on Linux is also not setuid root, so ReaLLite could have been modified further to read the process information directly from the /proc directory rather than through ps.

The critical file list and setuid root file lists were re-generated by the ReaLLite system since they differ significantly from Solaris to Linux. Table 8.1 details the changes made to the UNIX commands and inode data monitored. The format of the data provided by ps and netstat was also slightly different, but was handled by changing the awk commands that the data was piped through.

Information	ReaLLite Solaris Command	ReaLLite Linux Command
TCP RSTS Sent	netstat -s -P tcp | grep tcpOutRsts	netstat -s | grep 'resets sent'
Process Data	ps -ef -o ruser,user,pid, ...	ps ax -o ruser,user,pid, ...
Connection Data	netstat -n -P tcp	netstat -an
Location of ps	/usr/bin/ps	/bin/ps
Location of su	/usr/bin/su	/bin/su
Su log file	/var/adm/sulog	/var/log/auth.log

Table 8.1: Modifications made to ReaLLite for use on Linux

8.2 Testing of Linux ReaLLite

Linux ReaLLite was tested with a port scan attack and a SYN Flood attack. The port scan attack was launched from a remote host running nmap, as described in Chapter 7. The SYN Flood used is also described in Chapter 7. Linux ReaLLite identified both attacks immediately. Figures 8-1 and 8-2 show the output of Linux ReaLLite from both attacks. ReaLLite was also tested to determine if the inode data was accessed correctly. It was found that ReaLLite accurately reported when a user used the su command to obtain root-level privileges. Additionally, the system was run on a workstation for several days without false alarming. The performance of ReaLLite on the Linux Operating System closely parallels its performance on Solaris.

```
playground:~/reallite> uname -a
Linux playground 2.4.4 #11 Sat Apr 28 12:45:46 EDT 2001 i686 unknown
playground:~/reallite> ./reaLLite
990293665 [0] reaLLite ids starting up now
990293665 [1] new connection (or attempt) detected
990293665 [1] missed a process
990293665 [1] first layer scan triggered
990293665 [1] first layer scan completed
990293692 [9] SYN FLOOD DETECTED WITH 100 SYN_RCVD's
```

Figure 8-1: Linux ReaLLite detecting a SYN Flood attack

```
playground:~/reallite> uname -a
Linux playground 2.4.4 #11 Sat Apr 28 12:45:46 EDT 2001 i686 unknown
playground:~/reallite> ./reaLLite
990293544 [0] reaLLite ids starting up now
990293545 [1] new connection (or attempt) detected
990293545 [1] missed a process
990293545 [1] first layer scan triggered
990293545 [1] first layer scan completed
990293564 [9] PORTSCAN DETECTED WITH 1479 NEW RESETS
```

Figure 8-2: Linux ReaLLite detecting an nmap port scan

Chapter 9

Conclusions

9.1 Detection of Buffer Overflow Attacks

The detection of buffer overflow attacks typical of novice attackers is excellent. These attacks, found on hacker web sites, almost exclusively launch a root shell, which is easily detectable. ReaLLite detected buffer overflows that did not spawn root shells as well. Those that did not spawn root shells performed other malicious activities such as changing the permissions or contents of a critical file on the system. ReaLLite consistently detected all of these attacks as well.

With detailed knowledge about ReaLLite's data monitoring techniques, a highly skilled attacker could evade the system. Through delaying the execution of code after a buffer overflow, for example, an intruder could hide the attack from the IDS. Another stealthy attack could effectively "fake" a legitimate su to root after a buffer overflow.

9.2 Detection of Probes and DoS Attacks

With respect to DoS attacks such as SYN Floods and process table exhaustion, ReaLLite detected the attacks but could not gather much information about them. ReaLLite could see that a SYN Flood was happening, but could not report what service was being flooded or where the packets appeared to be coming from. Similarly with the process table attack, ReaLLite could see that it was occurring but was not able to report on where the attack was launched from.

The Nmap scan was also detected, but ReaLLite could not determine what ports were being scanned or what IP address the scan appeared to be coming from.

9.3 Overall Performance

The overall performance of the ReaLLite system was very encouraging. ReaLLite successfully detected a very wide variety of attacks without false alarming. ReaLLite had a very low probability of false alarming, however there was a high possibility of missing stealthy and evasive attacks. Normal attacks from hacker web sites, however, were reliably detected.

Because ReaLLite examines the system while an attack is taking place, signatures are not needed. This allows ReaLLite to detect novel buffer overflows. This is a very good feature since buffer overflow attacks are among the most common. According to [12], there have been eight new Solaris vulnerabilities discovered within the past three months. Out of the eight new vulnerabilities, five are buffer over attacks that ReaLLite is capable of detecting. Two other vulnerabilities exploit symbolic link relationships that ReaLLite may detect without modification. These attacks access setuid root programs directly before modifying a critical file, similar to buffer overflow attacks.

While ReaLLite does not detect all attacks, it does raise the bar for attackers. ReaLLite was very successful against buffer overflow, DoS, and port scan attacks. Several attacks are still able to get by the system, however. These attacks include stealthy buffer overflow attacks, slow port scans, stolen or sniffed passwords, and software misconfigurations. ReaLLite does not provide a complete security solution, but does offer good detection for an important class of attacks with little overhead, no kernel modules or audit data, and no associated security risks.

Chapter 10

Future Plans

10.1 Improvements in Detection

While ReaLLite did an excellent job of detecting the attacks it was tested with, further testing could help reveal weaknesses or problems. ReaLLite examined network traffic on approximately a dozen machines over the period of a few weeks, but that traffic is not indicative of normal network traffic since it was behind a firewall. More testing with a "live" Internet host machine could help strengthen the system.

10.1.1 Incorporate Netstat Into ReaLLite Code

One problem in moving ReaLLite across platforms is that standard UNIX commands, such as netstat, might have different parameters or formatted outputs. Another problem with the system is that each time it cycles, it uses many processes. ReaLLite must call ps and netstat, parse their output, and then perhaps call some other processes. Since netstat does not run as setuid root in Solaris, Linux, or other UNIX systems, it is possible to incorporate the netstat source directly into the ReaLLite system. This would solve the problem of different parameters and formatted outputs as ReaLLite would be able to get the network data directly without calling netstat as a separate process and having to parse its output. The speed of ReaLLite would increase as it would avoid having to fork so many processes. A user on the system might be able to notice ReaLLite running now because of the number of processes it creates.

Some UNIX Operating Systems, such as Linux, allow users to directly access the information about all system processes. On these systems, ps does not need to run as setuid root either. The ps source could thus also be incorporated directly into ReaLLite for Linux, which would increase performance of the IDS.

10.1.2 Hashes to Ensure Integrity

ReaLLite could make use of MD5 or SHA1 cryptographic hashes in order to ensure the integrity of the files which it uses and monitors. A small database of the correct hashes of the setuid root programs, the ReaLLite configuration and executable files, and some critical system files would help ReaLLite notice some stealthy attacks. For example, if an intruder knew that ReaLLite was running, he could modify the configuration files or the program itself. Using hashes to ensure the integrity of the IDS and the files it modifies would be very helpful and not add too much overhead. These hashes would not have to be computed regularly for all files but could be performed as needed, such as when a setuid root program was called.

10.2 ReaLLite Ported To Other Platforms

Bringing ReaLLite to other platforms would also be very useful. This would allow ReaLLite to be run on many more systems, and thus be tested against a wider range of attacks. By testing ReaLLite against FreeBSD attacks, for instance, the system may be modified in such a way that it is beneficial to the detection of attacks on other operating systems.

Porting to other platforms will also cause some problems that will need to be resolved. Although very similar, UNIX platforms always differ slightly. On OpenBSD the process ID numbers are not incremented by one for each new process as they are in Solaris, Linux, and other operating systems. Instead, they are picked randomly. ReaLLite would have to be modified because it relies on the process ID number to determine whether it has missed a process.

Another advantage of going cross platform is that some platforms allow for greater flexibility. As mentioned above, Linux does not require that you be the super user to read process data for all the processes on the system. ReaLLite for Linux could thus be optimized to access this data directly from the /proc directory. Similar optimizations could be made for other operating systems.

10.3 Further System Enhancements

ReaLLite could be extended to provide more accurate detection of a wider range of attacks and to be easier to set up and use. Some ideas for further enhancements include:

1. Randomize the name of the ReaLLite process running, or rename it to look like a system daemon. This would help hide ReaLLite from malicious users.

2. Set up a "ReaLLite Monitoring System" on another host that can monitor, and verify, that ReaLLite processes are running on different systems. Each ReaLLite host system could send a small ICMP packet to serve as a heartbeat, letting the Monitoring System know that the IDS is alive and well. The ICMP packets could contain some sort of counter, or encrypted payload, to verify that they are not being spoofed by an attacker.

3. Some exploit code uses the temporary directory, /tmp, to set up the attack, or store programs. Sometimes a buffer overflow attack will create a setuid root program in this directory, or a race condition exploit will create symbolic links there. ReaLLite could monitor /tmp to verify that there are no setuid root programs or other suspicious files or symbolic links present.

4. The netstat programs provides an immense amount of information, very little of which is currently used by ReaLLite. ReaLLite could be modified to read and analyze much more of this data.

5. The ReaLLite configuration and training could be more fully automated. ReaLLite should have a "training mode" where it gathers the information needed to set accurate thresholds for DoS attacks, and makes recommendations. Parameters should also be more flexible, so that they can be changed by command line options.

6. Scan for execution of known attacker tools such as nmap, sniffers, and password crackers. This would not just examine the name of the process, but the size and cpu usage to confirm that it is the suspected tool.

7. Keep private copies of netstat and ps to prevent the processes ReaLLite forks from updating the inode data and masking buffer overflow attacks. Hashes could be used to make sure that the private copies have not been modified by an attacker.

8. Incorporate some signatures from other IDS's for detection of more attacks. Snort [22] is a network-based IDS that uses signatures to detect all types of attacks, including probes, DoS attacks, and backdoors. The signatures that can be implemented using data provided by netstat could be added to the ReaLLite system in order to detect a wider range of attacks.

9. Create a user-friendly front-end, or GUI, for ReaLLite. ReaLLite is currently a command-line program that writes data to the screen. This output can then be piped into a file. A user-friendly front-end might allow for easier viewing of events in real-time, as well as extending logging capabilities.

10.4 A Usable Product

Developing this system to the point that it is a usable product for anyone would be very helpful. ReaLLite was coded as a proof-of-concept, but has shown that it can accurately detect many different types of attacks with no overhead. Properly developed, an IDS such as ReaLLite could run on anyone's desktop and increase security with no drawbacks. As a user process, it does not add any security threats to the host system it runs on, and its lightweight nature will not detract from the performance of the workstation or server it runs upon.

Bibliography

[1] The ABC's of IDS's (Intrusion Detection Systems). Carolyn Meinel. http://www.messageq.com/security/meinel_2.html. May 2001.

[2] BugTraq Mailing List Archive. http://www.securityfocus.com/bugtraq/archive. May 2001.

[3] CERT/CC Web Site. http://www.cert.org. May 2001.

[4] David A. Wheeler. Secure Programming for Linux and UNIX HOWTO. Chapter 8. http://www.linuxdoc.org/HOWTO/Secure-Programs-HOWTO/control-formatting.html. May 2001.

[5] Wietse Venema and Dan Farmer. The Coroner's Toolkit. http://www.fish.com/tct/. May 2001.

[6] Robert K. Cunningham, Richard P. Lippmann, David Kassay, Seth E. Webster, and Marc A. Zissman. Host-based Bottleneck Verification Efficiently Detects Novel Computer Attacks. *MIT Lincoln Laboratory*, 1999.

[7] CyberCop Web site. http://www.cybercop.co.uk. May 2001.

[8] Kumar J. Das. Attack Development for Intrusion Detection Evaluation. Master's thesis, Massachusetts Institute of Technology, June 2000.

[9] EMERALD eXpert-BSM Web site. http://www.sdl.sri.com/projects/emerald/releases/eXpert-BSM/. May 2001.

[10] J. W. Haines, R.P. Lippmann, D.J. Fried, M.A. Zissman, E. Tran, and S.B. Boswell. 1999 DARPA Intrusion Detection Evaluation: Design and Procedures. Technical Report 1062, MIT Lincoln Laboratory, 2001.

[11] Horizon, "Defeating Sniffers and Intrusion Detection Systems", Phrack, Vol. 8, Issue 54, File 10 of 12. http://packetstorm.securify.com/mag/phrack/phrack54/P54-10.

[12] ICAT Metabase, Computer Security Division at the National Institute of Standards and Technology. http://icat.nist.gov. May 2001.

[13] AXENT Technologies, Inc. Web site. http://www.axent.com. May 2001.

[14] Kathleen A. Jackson. Intrusion Detection System (IDS) Product Survey. *Distributed Knowledge Systems Team, Computer Research and Applications Group, Computing Information, and Communications Division, Los Alamos, New Mexico*, June 1999.

[15] Linux Intrusion Detection System (LIDS) Web site. http://www.lids.org. May 2001.

[16] Pars Mutaf. Defending against a Denial-of-Service Attack on TCP. *Department of Computer Engineering, Izmir Institute of Technology*, 1999.

[17] Intrusion Detection, Take Two. Greg Shipley. Network Computing. http://www.networkcomputing.com/1023/1023f1.html. May 2001.

[18] The Network Mapper (nmap) Web Site. http://www.insecure.org/nmap. May 2001.

[19] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. *Secure Networks, Inc. Report.*, January 1998.

[20] CERT Advisory CA-1999-16. http://www.cert.org/advisories/CA-1999-16.html. December 14, 1999.

[21] SecurityFocus Web Site. http://www.securityfocus.com. May 2001.

[22] Snort - The Open Source Network Intrusion Detection System. http://www.snort.org. May 2001.

[23] Aleph One, "Smashing the Stack for Fun and Profit", Phrack, Vol. 7, Issue 49, File 14 of 16, http://packetstorm.securify.com/mag/phrack/phrack49/P49-14.

[24] Sun's Solaris Web Site. http://www.sun.com/solaris. May 2001.

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

[26] Tripwire, Inc. Web site. http://www.tripwire.com. May 2001.

[27] Seth E. Webster. The Development and Analysis of Intrusion Detection Algorithms. Master's thesis, Massachusetts Institute of Technology, June 1998.