A STUDY OF ACCOUNTABILITY

IN OPERATING SYSTEMS

by

LEI ZENG

YANG XIAO, COMMITTEE CHAIR

HUI CHEN, COMMITTEE CO-CHAIR
SUSAN VRBSKY, COMMITTEE MEMBER
XIAOYAN HONG, COMMITTEE MEMBER
JINGYUAN ZHANG, COMMITTEE MEMBER
SHAN ZHAO, COMMITTEE MEMBER

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2014

ABSTRACT

Logging has become a fundamental feature within the modern operating systems due to the fact that logging may be used through a variety of applications and fashion.
Syslog daemon is the logging implementation in Unix/Linux platforms, while Windows Event Log is the logging implementation in Microsoft Windows platforms. These logging implementations provide APIs that in turn, simplify logging functions from data collection to data storage.

First, we introduce accountable administration. Accountability implies that entities should be held responsible for their actions or behaviors so that the entities are part of larger chains of accountability. Many security models and systems are built upon the assumption that super users are trustworthy. However, it becomes challenging to hold super users accountable since they can erase any trace of their activities. This chapter proposes an accountable administration model for operating systems where all system administrators can be accounted for even if they are untrustworthy. The accountability policy and operating system primitives are designed and constructed so that the proposed model is provable.

Second, Flow-net model is introduced in order to achieve better accountability, which means a logging system should be capable of capturing activities as well as the relationships among activities. Existing logging techniques record isolated events and rely on attributes and time stamps to establish their relationships, and this leads to probable loss of event relationships among large and complex logs. In this chapter, we present the design of flow-net methodology

and its implementation in current operating system such as Linux. We demonstrate that the flow-net logging technique is capable of preserving event relationships.

Finally, we leverage the overhead introduced by Linux Auditing Framework. Logging is a critical component of Linux auditing. The experiments indicate that logging overhead can be significant. The chapter aims to leverage the performance overhead introduced by Linux Audit Framework under various usage patterns. The study on the problem leads an adaptive audit logging mechanism. Many security incidents or other important events are often accompanied with precursory events. We identify important precursory events – the vital signs of system activity and the audit events that must be recorded. We then design an adaptive auditing mechanism that increases or reduces the type of events collected and the frequency of events collected based upon the online analysis of the vital sign events.

DEDICATION

This thesis is dedicated to everyone who helped me through this manuscript. In particular, Dr. Yang Xiao and Dr. Hui Chen who spent a lot of time supervising me finishing this dissertation.

ACKNOWLEDGMENTS

I would like to thank all of my committee members, Yang Xiao, Hui Chen, Susan Vrbsky, Xiaoyan Hong, Jingyuan Zhang, Shan Zhao for their comments and inspiring questions.

This work is supported in part by The U.S. National Science Foundation (NSF), under grants: CNS-0716211, CCF-0829827, CNS-0737325, and CNS-1059265.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer systems need to log data that represents system actions and the participating user activities. The former is largely due to the fact that logging is an essential entity for a variety of applications in computer systems. Logging implementations are logging facilities that facilitate applications to log data in different operating systems. Logging implementations may be system-specific or not but suffer with common performance concerns such as memory, disks, and CPU overhead. Logged data is often referred to as logs, audit trails/logs, or sometimes, traces. Logging can be performed by operating systems or by applications. Different information may be recorded based on different debugging or intrusion detection. Confidentiality and integrity of the logged data should be guaranteed, especially for digital forensics. Because of the large volume of logged data, filtering and analyzing the logged data draw many efforts. The logged data can be used by applications, such as auditing module, intrusion detection systems, or digital forensics applications and by persons such as system administrators or program developers.

System administrators may use logged data to diagnose system problems and to optimize system performance [1]. Logged data often contains large amount of system activities, which can be used to tune systems. Since the logged data often contains redundant information, efforts have been made to filter the logged data according to a specific analysis goal and thus reduce the analysis complexity [11].

Developers may use logged data to detect and correct program faults [5]. In this case, the logged data is often referred to as trace data or trace. Developers tend to use program traces to debug [35] since it contains the most detailed description of dynamic program behaviors.

Moreover, logged data can be used for auditing and logging becomes a requirement part in trusted computer systems [9, 10]. The logged data contains important information about the system activities, such as system call invocation, system resources usage, file access and users' behaviors. Auditing records include identification and authorization information which must be stored in computers securely [6].

In this dissertation, we will study the existing logging and accountability technologies in operating systems. We also explore the drawbacks of Linux logging and auditing and try to address some problems using logging technologies.

The rest of this dissertation is organized as follows. Chapter 2 gives brief overviews of operating system logging. Chapter 3 is to design an accountable administration in Linux operating system. Chapter 4 is to implement Flow-net methodology in Linux operating system. Chapter 5 is to explore the overhead and possible adaptation of Linux auditing. Conclusions of this dissertation are summarized in Chapter 6.

CHAPTER 2

RELATED WORK

2.1 Overview of Operating System Logging

Computer systems need to log data that represents system actions and the participating user activities. The former is largely due to the fact that logging is an essential entity for a variety of applications in computer systems. M=Many standards and regulations, such as PCI DSS [44], FISMlA [45], HIPAA [46], ISO27001 [43], COBIT [47], ITIL [48], CAPP/EAL4+ [49], LSPP [50], RBAC [51], NISPOM [52], and DCID 6/9 [53] mandate logging mechanism requirements. An event that occurred, should be logged with all necessary information, such as the performer of the event, timestamp, the name of the event, the object that the event operates on, the tool the event performed with, and the status of that event, as described in Fig 2.1, which is partially obtained from [17]. Logging implementations are logging facilities that facilitate applications to log data in different operating systems. Logging implementations may be system-specific or not but suffer with common performance concerns such as memory, disks, and CPU overhead. Logged data is often referred to as logs, audit trails/logs, or sometimes, traces. Logging can be performed by operating systems or by applications. Different information may be recorded based on different debugging or intrusion detection. Confidentiality and integrity of the logged data should be guaranteed, especially for digital forensics. Because of the large volume of logged data, filtering and analyzing the logged data draw many efforts. The logged data can be used by applications, such as auditing module, intrusion detection systems, or digital forensics applications and by persons such as system administrators or program developers.

System administrators may use logged data to diagnose system problems and to optimize system performance [1]. Logged data often contains large amount of system activities, which can be used to tune systems. Since the logged data often contains redundant information, efforts have been made to filter the logged data according to a specific analysis goal and thus reduce the analysis complexity [11]. Logged data in concise format without loss of information is proposed to enable easy interpretation and isolation of problems [30].

Developers may use logged data to detect and correct program faults [5]. In this case, the logged data is often referred to as trace data or trace. Developers tend to use program traces to debug [35] since it contains the most detailed description of dynamic program behaviors. Therefore, attentions on gathering and analyzing program executing traces have been drawn to optimize programs, diagnose failures, and collect program matrices [31-33].

Moreover, logged data can be used for auditing and logging becomes a requirement part in trusted computer systems [9, 10]. The logged data contains important information about the system activities, such as system call invocation, system resources usage, file access and users' behaviors. Auditing records include identification and authorization information which must be stored in computers securely [6].

Logging is also a basic requirement for implementing intrusion detection systems [7, 8]. The intrusion detection systems (IDS) are devices that monitor network or host activities for malicious actions or policy violations [34]. The IDS systems will maintain logged data that record system activities and check the logged data for suspicious activities in real time using predefined patterns. The decisions made by the IDS systems according to predefined policies are

often logged as well. Moreover, the IDS can post-process the logged data and generate decisions to its own log files.

The last application based on logged data is digital forensics. Digital forensics identifies the perpetrator and the criminal method by analyzing the logged data generated in computers [12]. Logged data records the activities of the computer, and by reading it; one may know what sort of event happened at any given, specific time. Therefore, integrity features of the logged data must be guaranteed. Most of the efforts that secure the logged data assume system administrators are trustworthy. In this case, the integrity of the logged data may be achieved through the use of kernel security modules as well as virtualization [12]. However, the administrator can modify the logged data and even modify the security rules that insure the security of the logged data. A paper [13] proposed an accountable administration approach that hardcoded protection logic of logged data in Linux kernel. In this case, as long as the kernel is loaded, system administrators cannot modify logged data in the system.

For the first part of this chapter, we provide a survey focusing on general requirements for logging mechanisms highlighted in some regulations and standards. Then we survey IETF Syslog Protocol that is widely used for Unix/Linux logging and logging implementations such as Syslog daemon and Syslog-ng on Unix/Linux platforms. Next, we explore the logging implementation in Microsoft Windows platforms. Then we discuss some concerns regarding logging mechanisms and logging applications. The last part is the future work.

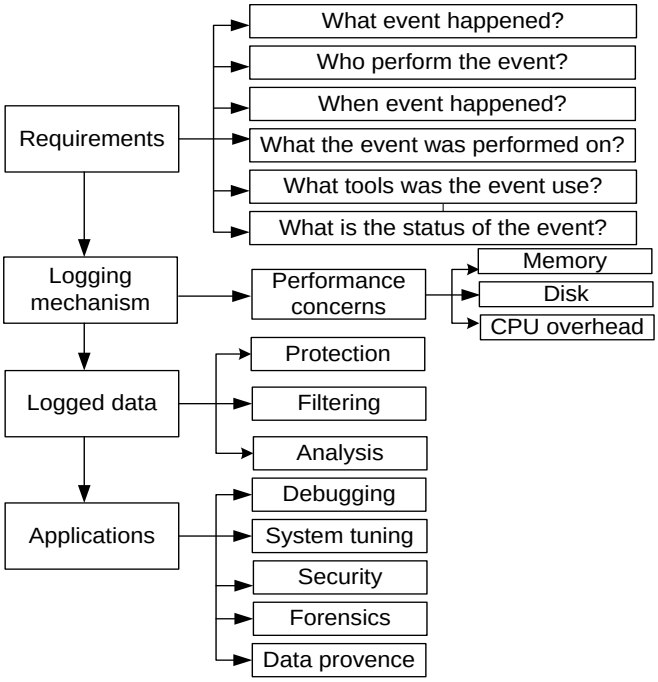

Fig 2.1. The logging diagram

2.2 Logging Mechanism

Logged data and tools to manage it have been critical in the past few years. Log management has emerged as a fundamental feature within information systems, driven not only by compliance mandates and elusive threads, but also by increasing complexity of IT environments [18]. Currently, many regulations, such as PCI DSS and ISO27001 mandate the log management technology, from log collection to logged data retention [18]. Overall, these regulations have similar requirements in terms of log management.

2.2.1 What Should Be Logged

The following discussion will focus on ISO27001. Issues identified in 2006 SANS summit include the difficulty of generating appropriate logged data and integrating the logged data with corresponding management functions [17]. In ISO27000 family, it mandates activities that should be logged, as described in Fig 2.2 [18]. The left side in Fig 2 describes logging categories that are mandated in ISO27001, while the right side depicts their corresponding purposes. First, authentication, authorization and access events should be recorded. Successful and failed authentication, as well as access to applications or data, may provide useful information for tracking and investigating access to systems by attackers as well as authorized users [18]. Secondly, system or application changes (especially privilege changes) may cause exposure and subjection to attacks. Sometimes, these alterations are made from malicious insiders or attackers [18]. Subsequently, availability issues such as application modules and systems should be logged as well. Faults and errors may affect the availability of applications and even compromise data [18]. Next, detecting resource issues out of security and operational concerns is a common operation. Therefore, resource issues such as exhausted resources and network connectivity should be logged. Fifth, known threads that are logged are of significance, especially for intrusion detection systems (IDS) that need to detect and block attacks if necessary [18].

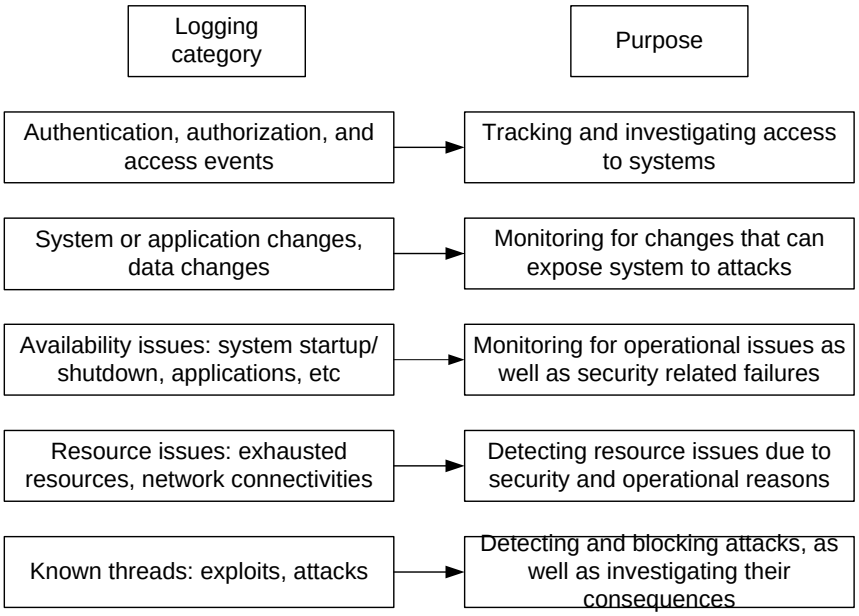

Fig. 2.2. Mandate logging categories with corresponding purposes [18]

2.2.2 Elements of Logged Data

Necessary elements in logged data are identified in Table 2.1 [17]. No matter how the logged data is organized, it should at least contain 7 elements: 1) type of action; 2) subsystem that performs the action; 3) identifier for the performer; 4) identifier for the object that the action performed on; 5) before and after value; 6) date and time; 7) result of the action. Note that the identifier for the action performer and the object that an action performed on should be as many as available [17]. Identifiers may contain unique identifier of records accessed in database, computer name, or IP address [17].

Table 2.1 Elements of the Log and Examples [17]

Elements of the log	**Examples**
Type of action	Create and update files
Subsystem performing the action	Process or transaction name
Identifier for subject(performer)	Computer names and IP address

Identifier for object(action performed on)	Computer names and IP address
Before and after value	When update data element
Date and time	Timestamp of the action
Result of the action	Whether action is allowed

2.2.3 Retention of Logged Data

Different standards may have a variation of retention policies. Remember that retention policies are users' requirements and related to available resources of a computer system. In the following part, we will review the retention requirements of different standards, such as ISO27001 and PCI DSS.

In section 10.10.5 in ISO27001, "action to be taken" requirements imply not only collecting logged data but also defining follow up procedures such as investigating and monitoring logged data [18]. ISO27002 requires that "procedures for monitoring use of information processing facilities shall be established and the result of the monitoring activities reviewed regularly [19]." ISO does imply follow up procedures considering the reviewing of the logged data. For instance, such a reviewing procedure should include periodic or real time logged data review workflow on application basis. Besides, ISO27001 states, "logging facilities and log information shall be protected against tampering and unauthorized access [18]." Protection includes preserving confidentiality and integrity of logged data. In regard to confidentiality, role based access control with least privileges or mandatory access control may be enforced to protect logged data [18]. In regard to integrity, the logged data can be written to "write once" devices such as DVD. The standard implies that malicious insiders should not be

able to modify logged data to conceal their attacks. Note that loss of logged data due to lack of storage capacity is not tolerated, stated in the standard [18].

However, in section 3.1 in PCI DSS, protect the data for cardholder in payment systems are required [44].

2.2.4 Logging in Virtualized Environment

Virtualization makes it possible to run multiple systems in a single hardware platform and is a trend within the modern information industry. What happens to logging management in virtualized environments?

Firstly, let's review the common parts for logging management in virtualized environments: 1) Audit logging shall be performed across all physical and virtual components; 2) The activities of the administrator shall be logged across all physical and virtual environments; 3) Procedure of monitoring and reviewing the logged data shall cover both physical and virtual environments; 4) Logged data shall be protected no matter if it is produced in physical or virtual environments; 5) Time synchronization shall be guaranteed across all log sources [18]. In essence, there is no primary reason to rid away logging tools that work for physical environments. Given the fact that a mix of physical and virtual environments is the trend in IT, logging technology in a physical environment will continue to help and deliver value [18].

Second, let us discuss the different parts for logging due to the introduction of virtualization: 1) Logging from guest operating systems as well as virtual networking modules; 2) logging the actions of VM administrators; 3) defining the process of reviewing and monitoring

the logged data across all layers, including identifying attacks against VM infrastructures; 4) distilling logged data from transient VMs where logged data cannot be protected; 5) synchronizing time for both guest operating systems and possible components in clouds [18]. Due to the introduction of virtualization, a lot of new technologies have been brought in. In the meanwhile, these new technologies may incur new problems. Therefore, the changes for logging in virtualized environments must be understood to better develop a compliance system.

2.2.5 Storage and format

The system must support the storage and format of logged data so that integrity of logged data will be guaranteed and analysis and reporting tools will be supported [17]. Mechanisms that support these goals include: 1) Microsoft Windows Event Logs; 2) Logged data in well documented format collected by syslog, syslog-ng or syslog-reliable network protocols; 3) Other logging mechanisms that meet above identified requirements [17].

We have finished general discussions on logging requirements highlighted in the common criteria and in other literatures. Now we can discuss how these requirements are translated to a particular operating system implementation, such as syslog daemon and its next generation version syslog-ng in Unix/Linux platforms, as well as Windows Event Log in Microsoft Windows platforms.

2.3 Logging in Linux

Many operating systems have provided logging facilities in their respective kernels. The logging facilities in these operating systems have become essential components of the kernels. Logging can be an add-on to an operating system. In this case logging is not shipped with the operating system kernel. Our study on Linux systems is based on Linux kernel 2.6(http://www.kernel.org). Linux has two levels of operating system level logging: kernel level and non-kernel level.

Table 2.3.1 Elements of the Log and Examples [17]

Priority	**Descriptions**
Emergency	System is unusable
Alert	Action must be taken immediately
Critical	Critical conditions
Error	Error conditions
Warning	Warning conditions
Notice	Normal but significant condition
Info	Informational messages
Debug	Debug-level messages

2.3.1 IETF Syslog Protocol

Syslog [2][3][4] is the most widely used protocol for Unix/Linux logging. It allows separation of applications that produces logs from the systems that transmits and stores them and applications that generate analysis report [27]. It is supported by a variety of devices in different platforms and can be used for security management as well as analysis and debugging [27]. Messages from facilities (showed in Table 2.3.1) are assigned with priority levels (as showed in Table 2.3.2).

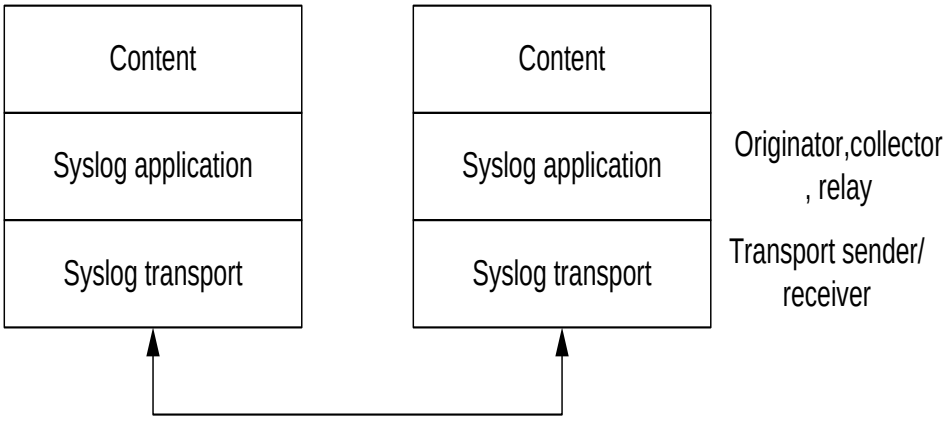

Fig. 2.3 Syslog layer [29]

Syslog utilizes three layers: Syslog content, Syslog application and Syslog transport, as shown in Fig 2.3. The Syslog content layer contains management information in a Syslog message,; the Syslog application layer is responsible for handling generation, interpretation, routing, and storage of the message [29]. There are three functions in the Syslog application layer: originator, collector, and relay. An “originator” is a device that generates the message. A “collector” is a device that collects the message. A “relay” is a device between the originator and collector, forwarding messages to another relay or collector. A transport sender or receiver is on the Syslog transport layer, which passes messages to a specific transport protocol or gathers messages from a specific protocol [29]. Syslog packages are sent using User Datagram Protocol (UDP) and, thus, is not reliable.

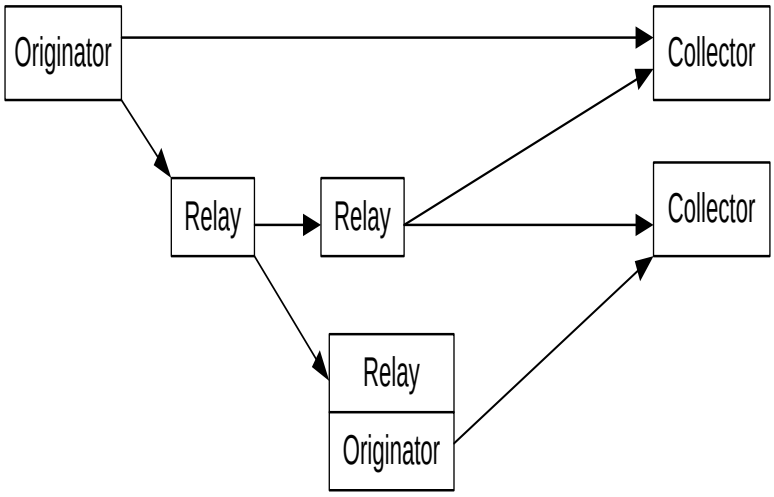

Fig. 2.4. Example deployment scenarios [29]

Example: deployment scenarios are shown in Fig 3. The boxes in Fig 4 represent syslog-enabled applications. Note that relays may send some or all of the messages that they gather and may generate their own messages to send. In the scenario shown in Fig 2.4, messages may be sent directly from originator to collector or through some intermediate relays to the collector. One single message can be sent to multiple collectors. Sometimes, relays may become originators and generate their own messages to send.

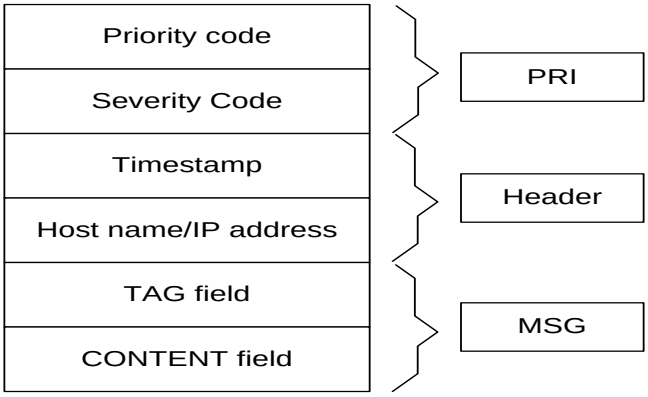

Fig. 2.5. Format of a Syslog Packet [28]

A syslog packet contains three distinct parts: PRI, HEADER and MSG, as shown in Fig 2.5. The PRI part represents both the severity and facility of the message. The timestamp in the HEADER part is the time that the message which generates the message. The MSG field fills the rest of the packet and contains the tag field and content field. Note that the MSG part is a free format that contains the contents of the event.

2.3.1.1 Some issues in Syslog Protocol

First, as mentioned above, UDP protocol is used to transmit Syslog packages, and thus, the transportation is unreliable. The messages sent through the network may be dropped because of network congestion or discarded by a malicious insider on purpose.

Second, for simplicity, Syslog protocol is designed only for message transportation. There are no format requirements in the MSG portion which contains the contents of the messages. Therefore, different applications or operating systems have the ability to generate different formats for MSG parts, which will subsequently cause trouble when distilling valuable information from logged data generated from a variety of devices.

Third, since there is no authentication procedure in Syslog package transportation, a receiver that receives the message cannot ascertain that the message is indeed sent from the originator. Malicious users may generate a large amount of fake messages.

Furthermore, as mentioned above, malicious user may use fake messages that indicating problems on some machine. When administrators receive the fake messages, they may be

mislead to a wrong analysis road and at the same time malicious user may attack another machine.

Fifth, packets sent using Syslog protocol may be disordered. There is no mechanism to guarantee ordered package delivery.

Sixth, malicious users may record a large amount of packages that indicating normal operations in a specific machine and resend these messages when they attack this machine.

Seventh, false configuration of Syslog daemon may cause a machine receive packages sent by its own machine. There is no status field to indicate the sender status.

2.3.2 Kernel Message Buffer

Linux kernel emits kernel messages automatically, whenever deemed necessary. A kernel message can be regarded as a program trace (text/debug message) generated by Linux kernel. Kernel programmers insert statements that generate kernel messages. Usually, kernel emits kernel messages whenever an error happens to occur. These messages are primarily for debugging purposes. Therefore, logging kernel messages are not sufficient because non-complaining system calls seldom generate any messages, though they do provides some insights into kernel activity.

Linux maintains a “ring buffer” in its kernel. The ring buffer is a cyclic buffer and is currently implemented as a character array (defined in module kernel/printk.c). The size of the ring buffer must be a power of 2. By default, the size is 16K, 32K, 64K, or 128K bytes depending on the architecture of the processors. The ring buffer is referred to as the kernel

message buffer. In any context of Linux kernel, a kernel message can be inserted into the Linux ring buffer by calling Linux kernel routine printk (defined in module kernel/printk.c). The kernel routine also has the capability of printing out the message on a `Linux console under condition that there is not another routine is printing out on the console. The condition is ensured if a semaphore on the console (i.e., a mutex named console_sem defined in kernel/printk.c) is grabbed successfully. Since the ring buffer is a cyclic buffer, the oldest message will be overwritten when the buffer is full. The Linux kernel does not have any guarantee that the kernel message to be overwritten will be copies anywhere else since kernel routine print does nothing before it overwrites the oldest message. Therefore, it is actually impossible to preserve the kernel message in its completeness. In practice, a separate process can dump all the messages in the buffer to somewhere else. For example, a file, before the buffer is full, though this solution leads to a race condition. A dedicated kernel message buffer is subsequently deployed to guarantee all the messages in the dedicated message buffer are be written out to a log file prior t the message is overwritten, using a lock on the dedicated kernel message buffer [13].

The kernel messages may be accessed from outside the kernel. One can read a kernel message from /proc/kmsg or system call sys_syslog, though they are ultimately the same due to the fact that both of them use kernel routing and conduct_syslog to access the ring buffers. A fundamentally important difference exists between these two. Only root has read permission on file /proc/kmsg, while system call sys_syslog allows a non-root process read up to the last 4K bytes of messages from the ring buffer.

Linux has a utility program called dmesg that reads and prints out all messages in the ring buffer. Linux provides a daemon program to interact with the ring buffer, which is klogd. When a Linux system boots, an “init process” will start a klogd process. The daemon process monitors the kernel ring buffer and retrieves the content of the buffer. It often interfaces with another daemon service syslogd, which will be introduced next. Through daemon service syslogd, the kernel messages will be saved in /var/log/messages. Note that daemon service syslogd added time stamp and some textual information to show that the log messages were generated by the kernel. The time stamp is not actually when the event happens. It is the time when the messages are read from the ring buffer and saved in /var/log/messages.

2.3.3 Syslog daemon implementation

Syslogd and syslog-ng are the syslog daemons implementations in Linux operating systems. They not only can log data from their own machines but also log data from other machines [12]. Syslogd consists of two programs: klogd and syslogd (Fig 2.6). Klogd is responsible to manage logging in kernel space while syslogd manages logging in user space. Applications can generate their own logging files.

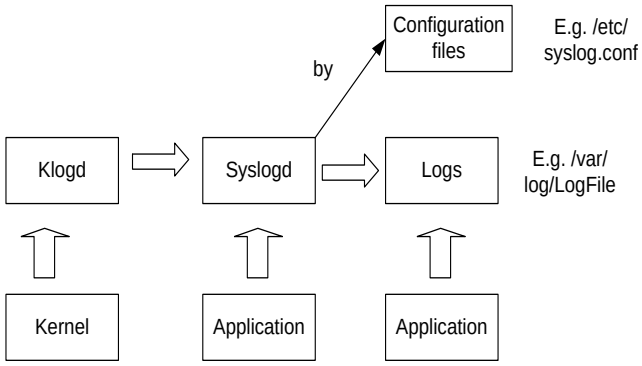

Fig. 2.6. The workflow of syslogd [12]

2.3.3.1 Syslogd

Linux provides a facility for assisting logging. This facility, as we have mentioned in the above, is a daemon service syslogd. Daemon service syslogd is a utility that intends to provide a uniform interface for logging. A logging message can be sent to the daemon by using a libc function syslog. However, whether a non-kernel component sends any logging messages to the service at all depends on the component itself. Excluding the time stamp, the host name, and the process information, which information is dumped in which format depends on Linux-PAM.

Daemon service syslogd listens to Unix domain socket /dev/log, from which local log messages are read. In fact, syslogd has the capability to be configured to listen to up to 19 additional sockets without changing its source code.

In addition, daemon service syslogd supports remote logging. Process syslogd in a host can forward received logging messages to another syslogd process running another host at port 514 by a UDP connection. One may4 configure syslogd by modifying its configuration file such

that designated logging messages will be forwarded to another syslogd process running at another host.

2.3.3.2 Syslog Configuration

Syslog has the ability to accept data from any local processes, from kernel through klogd, and may even derive data from processes on remote machines [36]. By configuring syslog, users can determine which data to log and where the logged data shall be stored. All Unix and Linux variants basically contain a preconfigured syslog. The syslog daemon, syslogd, process receive log messages based on message type (sometimes referred to as facility) or priority. By modifying /etc/rsyslog.d/50-default.conf (Ubuntu 11.04) file, users have capabilities of specifying the mapping actions to a specific facility and/or priority. Each line in the file contains facility/priority selector and corresponding action. For instance, "mail.info" is the facility/priority selector, and "/var/log/mail.info" is the corresponding action. In the example, "mail" is the facility and "info" is the priority level. Whenever syslog daemon receives the log messages from "mail" facility with the priority specified as "info", the corresponding action syslog daemon will store the message to "/var/log/mail.info."

Table 2.3.1 Syslog priority level [17]

Priority	**Descriptions**
Emergency	System is unusable
Alert	Action must be taken immediately
Critical	Critical conditions
Error	Error conditions
Warning	Warning conditions
Notice	Normal but significant condition
Info	Informational messages

Debug	Debug-level messages

The facilities defined in Linux are listed in Table 2.3.3. These facilities are often referred to as categories. The facilities have no relation to each other, while priorityes are classified hierarchically. The supported priorities in Linux are known as the following: emerg, alert, crit, err, warning, notice, info, and debug (in decreasing order of urgency). It is the programmer who determines the emergency of a given message, not syslog daemon. The program will set the facility and priority of generated messages.

Table 2.3.2 Elements of the log and examples [17]

Elements of the log	**Examples**
Type of action	Create and update files
Subsystem performing the action	Process or transaction name
Identifier for subject(performer)	Computer names and IP addresses
Identifier for object(action performed on)	Computer names and IP addresses
Before and after value	When update data element
Date and time	Timestamp of the action
Result of the action	Whether action is allowed

The most common action used in practice is to store the log messages to a specific file (full path is specified in /etc/rsyslog.d/50-default.conf in Ubuntu 11.04). If the specified file exists, the log messages will be appended to the file. If not, new file with the specified name will be created by syslog daemon. The log messages can be directed to a file, a named pipe, a device file, a remote host, user's screen, or discard. Named pipes are commonly used for debugging. TTYs are the device files that are often used. However, some programmers tend to send security messages to /dev/lp0, which is a local line printer. Note that messages that are printed out cannot be wiped out or modified by hackers, simultaneously while hackers using the compromised

administrator account may alter the log messages stored in files. With regard to the former notion, log messaged can be forwarded to a remote host (by default, syslog daemon do not accept remote messages). Note that the syslog daemon on remote host should be configured with "-r" flag to allow syslogd to receive messages from another machine. A central logserver may be deployed using the remote feature of syslog daemon. Any log messages that do not match any selectors in /etc/rsyslog.d/50-default.conf files will be discarded. Unexpected behavior may be incurred by contradicting selector-action mapping.

Table 2.3.3 Linux logging facilities [36]

Facility	**Descriptions**
Auth	Used for many security events
Authpri	Used for access control related messages
Daemon	Used by system processes and other daemons
Kern	Used for kernel messages
Mark	Messages generated by syslogd itself that contain only a timestamp and the string "--MARK--"
USER	The default facility when none is specified by an application or in a selector
Local7	Boot messages
*	Wildcard signifying "any facility"
None	Wildcard signifying "no facility"

2.3.3.3 Data that Can Be Logged

System memory in the Linux operating system can be divided into two distinct regions: kernel space and user space. Syslog application programming interface (API) cannot be used in kernel space because the kernel itself provides the API. There is a dedicated logging tool known as "printk" for kernel message logging. Since "printk" can be used anywhere in kernel code,

events as small as kernel functions can be logged using printk, as long as printk is placed before or after the function call. Therefore, the collected kernel functions are written to kernel ring buffer and the syslog daemon will write out the buffer periodically. However, since some of the kernel functions are written in Assembly language, there is no way to use printk between these code lines in Assembly language. Therefore, the use of these instructions in Assembly language cannot be trapped. One common issue is that if a trivial function event is trapped in kernel, users do not have sufficient information in regards to what events in larger scale happen because the trivial function event may be triggered in many larger events. Another primary issue would be the fact that the trivial function event is used in many larger events and trapping the trivial event may incur tremendous overhead because it is frequently used. Taking the scale of the events and system overhead into consideration, usually some system calls are trapped.

In user space, there are syslog APIs such as openlog, syslog, and closelog library calls available for programmers to log messages. When programmers need to collect shared libraries that are used in their applications, they may use syslog APIs to the place they refer to as the "shared library."

2.3.4 Syslog-ng implementation

Syslog-ng [14] is an implementation of the Syslog protocol in Unix/Linux systems. There are two editions of syslog-ng available: syslog-ng open source edition and syslog-ng premium edition. In this survey, we only cover syslog-ng open source edition. Moreover, syslog-ng [16], which represents the advanced version of syslog, is intended to replace syslog. In fact, Syslog

and Syslog-ng can inter-operate in a network environment. Both Syslog and Syslog-ng provide facility, levels, timestamp, etc. The major difference of Syslog-ng is its implementation of "filtering" of logging messages, using either TCP or UDP for transporting logging data in a network at users' discretion, encryption/decryption of transmitted messages in network, and interface with popular database systems.

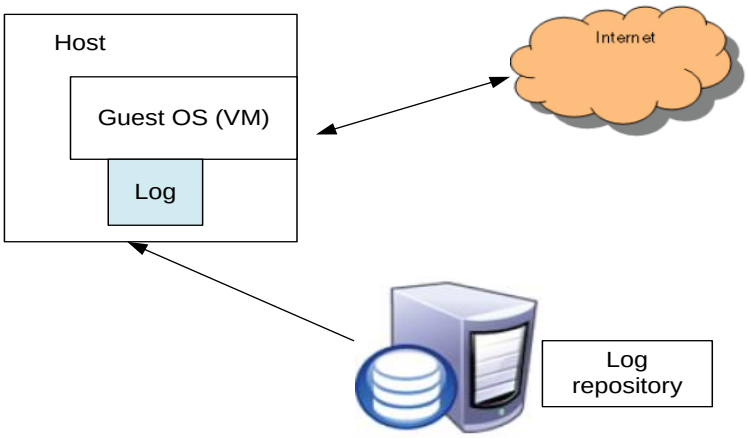

Fig. 2.7. Logging scheme [12]

Syslog-ng is part of many Unix/Linux distributions. Some distributions install it by default to replace original syslog daemon for logging, while others provide installation packages or upgrade paths. Syslog-ng tries to provide features that syslog daemon is lacking. The main benefits for syslog-ng are discussed as follows.

2.3.4.1 Powerful configurability

The syslog daemon syslog-ng is used by many services to log system events. The advantage over syslog daemon is that syslog-ng only uses one configuration file, while syslog

daemon accepts messages and processes received messages based on configuration files /etc/sysconfig/syslog and /etc/syslog-ng/syslog-ng.conf.

2.3.4.2 Filtering based on message content

Filters are boolean expressions that can be attached to a specific message with its value to be TRUE or FALSE. The use of these boolean expressions will promote the analysis performance.

2.3.4.3 Portability

The syslogd application is a standard system logging application used by a variety of devices such as routers and switches, as well as servers on operating systems based on Unix, including HP-UX, BSD, Solaris, AIX, and Linux but not Microsoft Windows. The implementation of syslogd is system specific, while syslog-ng uses the same code on different platforms.

2.3.4.4 Better network forwarding

Regarding reliability, syslogd uses UDP protocol to transmit messages, which is unreliable. In addition, syslogd simply drops messages if the collector is not available and does not support message encryption. The syslog-ng application using TCP can provide better reliability.

2.3.5 Logging data protection

The traditional way to protect logged data is to use “write-once” devices such as DVD [6]. As formerly mentioned, logged data that is printed out cannot be compromised by malicious users. However, the volume of the printed out data causes difficulty to analyze [5]. Stored logged data to “write-once” devices are not sufficient to guarantee the trustworthiness of their associated data, including structures and attributes, which are compromised [7].

Moreover, hash and digital signature are widely used for logged data protection. In [9] [10] [11], forward integrity MAC (Message Authentication Codes) were developed to protect logged data. However, the attackers may delete logged data. In [13], hardcoding logged data protection logic into kernel may prevent users, even administrators, to delete logged data. There are two other approaches to protect logged data: kernel module and virtualization [12].

Since an application will ultimately enter kernel space to access system resources, we can deploy a kernel module with protection logic to protect the integrity of logged data. The other approach is through virtualization, as shown in Fig 7. Virtualization allows multiple guest operating systems to be installed on a single hardware platform and presents those guest operating systems with virtualized resources. Since one of the most important features of virtualization is isolation of each virtual operating system, each operating system should function independently and cannot gain information from other devices other than virtual ones through guests sharing resources such as I/O devices, memory and CPU [12]. This isolation allows the host machine to function securely without attacks from guest operating systems through the Internet [12].

Now we compare these two approaches. Obviously, kernel module approach is better than the virtual machine approach in terms of efficiency [12]. However, virtualization approach is more secure than kernel approach [12]. Suppose an attacker hacked the administrator account and gained administrator authority [12]. The whole system deploying kernel module becomes in danger. However, the host operating system and other VMs will not be effected even if an administrator account in guest machines has been hacked [12].

Due to the fact that logged data transmission uses Syslog protocol and thus, do not guarantee package delivery, packages can be missing or tampered.

2.3.6 Logging in the Linux Auditing System

The Linux Auditing System [38-39] development starts from 2004. It has been shipped with version 2.6 of the Linux kernel. The Linux Auditing System contains and exploits two separate, major components, known as the kernel component and the user-space component. The design of the Linux Auditing System is based on the requirements defined in the Controlled Access Protection Profile (CAPP) found in the "Common Criteria for Information Technology Security Evaluation (CC) [10]" [38]. Supposedly, the auditing system helps Linux to reach C2 level security. The Linux auditing system was designed as a generic auditing framework. There are two ways to generate audit events: generating events in kernel space and generating events in user space [38-39].

The kernel level component attaches and audits context to each process. The audit information includes information that identifies the active process and the associated user, i.e.,

process id (pid), user id (uid), group id (gid), and so on. The information about the kernel objects effected by the system referred to as inter-process communication (IPC) objects, sockets, inodes, and so forth may be saved as auxiliary information. The Linux auditing system uses netlink socket (i.e., protocol family PF_NETLINK or address family AF_NETLINK) to communicate with user space processes. All the audit events are queued in the netlink socket in the kernel space. Using the netlink socket has a few advantages. First, the netlink socket routines are already in the core of the kernel. Thus, any module that uses the netlink socket does not have to be statically linked to the kernel like system calls do. Second, the netlink socket supports multicast, i.e., a process can send a message to a group of addresses through a netlink socket. Third, the netlink socket uses asynchronous communication. It can simply leave a message in the queue and the message can be processed some time later. This offers more flexibility for kernel job scheduling. However, this is also a disadvantage because the processes in the user space needs to periodically poll the socket to check if there is any status change. The auditing system divides all its message types into several blocks. Among them, message types 1000-1199 are bi-directional, message types 1200-1299 and 2100-2999 are exclusively user space, and message types 1300-2099 are for one way communication from the kernel space to the user space. Message types 1300-2099 are usually used to deliver audit events from the kernel space to the user space, for example, audit information on a system call can be delivered through message type 1300 (AUDIT_SYSCALL).

The kernel component thus adds not only system call auditing, but also auditing events smaller than system calls such as file access and network layer actions.

The user space component consists of a daemon program auditd and a few configuration and reporting tools. The auditd daemon interacts with the kernel auditing component and the user space processes. It can control kernel auditing behavior and store audit events in a file, by default, /var/log/audit/audit.log.

The system is implemented through directly adding function hooks into the kernel. A character special device /dev/audit is added and it provides a communication channel for auditing components. The user-space components have a daemon program, called auditd. This daemon program can disable or enable kernel system auditing, providing auditing ruleset through /dev/audit to the kernel components. The ruleset determines what system calls and when the system calls will be audited. The daemon program auditd also retrieves audit records through device /dev/auditd. The audit records will then be dumped to a log file. Just like any other UNIX I/O device, this special character file can be controlled by ioctl() system call. When the auditing function is off, the performance penalty introduced by the audit system is neglected. The audit records will be inserted in an audit event queue in main memory. Once the queue is full, no more audit events are allowed to generate. The events are retrieved by the audit daemon.

User applications can generate their own audit records for explicitly issuing the audit library function calls. This level of logging is a replacement and also an enhancement for the low-level system call based records. In general, they have higher granularity. However, only trusted applications can generate these kinds of events. Furthermore, when the user level events are generated, the kernel events (system call/file access) can be suppressed. For example, utility application “passwd” is used to change the password for a user. This application obviously

involves open files, read files, and close files. One can suppress auditing for those system calls while generating only one audit event for the password change in the user space.

2.3.7 Summary of Linux logging

For syslog daemon implementation, the kernel's console printing utility printk [14-15], whose semantics are the same as printf in C standard library, is the simplest logging tool in Linux. In kernel space, there is a ring buffer. Kernel components use the printk function to insert kernel messages into the ring buffer. The ring buffer can be read or cleared through the /proc file system, in this case, the /proc/kmsg file or the sys_syslog system call from the user space.

In the user space, daemon service klogd reads ring buffer messages through the /proc/kmsg file or the sys_syslog system call if the former is not present. Although daemon service klogd can dump messages to a file, more often than the former, interfaces with daemon service syslog;, which will, in turn, dump the message into the /var/log/messages file. Note that user space system components often send their logging messages to daemon service syslogd. In fact, the very purpose of daemon service syslogd is to serve as a concentrator for all logging messages, and it is not uncommon that many system or application level components such as Dynamic Host Configureation Protocol (DHCP) client, Network Time Protocol daemon (ntpd), and Linux-PAM (Pluggable Authentication Modules for Linux) send their log messages to it.

2.4 Logging in Microsoft Windows

Microsoft Windows platform for desktops and servers as well as workstations is widely used by organizations. Starting from Microsoft Windows Vista, Microsoft has redesigned its logging format. The new EVTX stores logged data as a binary stream in XML. Microsoft provides new application programming interface (API) that is no longer supported in older Windows platforms to access logged data. Besides, the structure and data within the fields in EVTX records has changed dramatically [20]. To support these changes, new Windows Event Logging framework that includes increasing functionality has been developed. Note that EVTX format is defined as a basis for Windows Event Logging framework.

Microsoft's EVTX log format will be covered first in this section, and then Windows Event Logging framework will be explored.

2.4.1 EVTX format

EVTX format was redesigned mainly because its original version EVT log format since Windows NT 4.0 has been there for a long time with few updates. New features in EVTX log format include: new event properties, an Extensible Markup Language (XML) format, the use of channels for publishing events, a new Event Viewer, and a revised Windows Event Log service [21].

2.4.1.1 EVTX Event Definition

EVTX supports several new event properties. Table 2.4.1 includes the most common event properties that are defined in Microsoft.

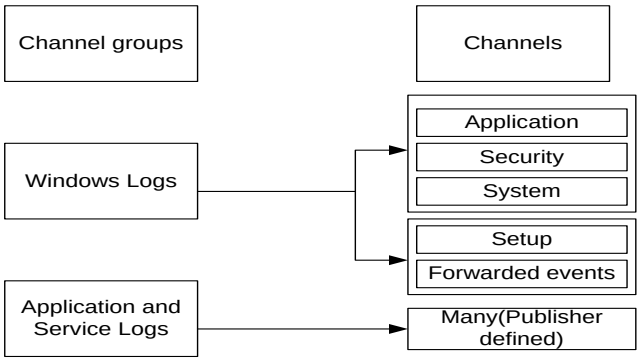

Fig. 2.8. EVTX channels [21]

One important change in EVTX implementation is that channels can be used to store events [21]. Channels are defined by Windows Event Log Software Developer Kit as streams of events that can be used by applications and operating systems to publish events to a log subscriber [21, 23]. The channels that are mainly used are divided into two groups: Windows Logs and the Application and Services Logs, as shown in Fig 2.8. The Windows Logs channel group includes system, security, and application channels as well as two new defined channels, Setup and Forwarded events channels [21]. The Application and Service Logs channel contains many channels that publish events from a single components or application [21].

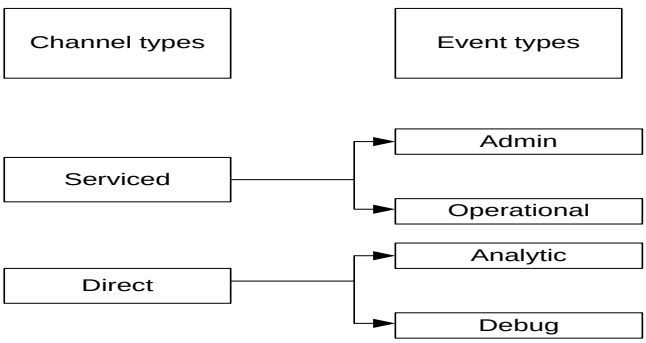

Fig 2.9. EVTX channel types and event types [21]

There are two channel types for each channel group and one event type for each event. There are two types of events, Admin and Operational, for a serviced channel type, as shown in Fig 2.9. The direct channel type includes two types of events, Analytic and Debug events [21]. The main difference between these two channel types is that direct channels cannot be forwarded and/or collected remotely while serviced channels can [21, 23].

As mentioned above, EVTX logged data is stored in XML format. Logged data in XML format can significantly increase the granularity that can be applied when any other third party software or Event Viewer tries to view the logged data [21].

Table 2.4.1 Logging properties and descriptions in Windows [22]

Property name	**Descriptions**
Source	The software that logged the event
Event ID	A number identifying event type
Level	A classification of the event security
User	The name of the user on whose behalf the event occurred
Operational Code	Numeric value identifying the activity
Log	The name of the log where the event was recorded
Task Category	Represent a subcomponent or activity of the event publisher
Keywords	Tag used for filtering and searching events
Computer	The name of the computer on which the event occurred
Date and Time	The date and time that the event was logged
Process ID	The identification number for the process
Thread ID	The identification number for the thread
Processor ID	The identification number for the processor
Session ID	The identification number for the session
Kernel Time	The elapsed execution time for kernel instruction
User Time	The elapsed execution time for user instruction
Processor Time	The elapsed execution time for user instruction

Correlation ID	Identifies the activity in the process for which the vent is involved

2.4.2 EVTX components

The EVTX format is called Windows Event Log, which contains a new Event Viewer and a modified Windows Event Log service [21].

2.4.2.1 Event Viewer

Event Viewer is a component of Windows Event Log framework and is used by users in Windows to view event logs locally or on a remote machine [24]. Some noticeable changes for the new Event Viewer include: 1) the ability to attach tasks to events; 2) advanced filtering based on XML; 3) the ability to use log subscription to gather events remotely [21]. The new Event Viewer has a graphic user interface to let users view logged data in XML, EVTX, TXT, and CSV format [21].

The ability of creating a custom view to filter the logged data is the most useful feature of the Event Viewer. This ability can dramatically reduce the amount of logged data that is displayed, significantly reducing the searching time to locate a particular event. The custom view window can allow end users to filter logged data based on timestamp, level, log, source, event ID, and computer. It also allows end users to edit XML query to filter the logged data if necessary. Besides, the Event Viewer provides an interface to attach a task to a specific event [21].

2.4.2.2 Windows Event Log Service

A previous version of the Microsoft Logging product was found to have scalability and performance issues [21]. Therefore, Windows Event Log service was revised to erase these drawbacks and is able to publish events in an asynchronous manner so that the event publisher does not need to wait for the Windows Event Log service to store the event [21, 25]. The Windows Event Log service will process the published events based on their types. Different types of events require different processing procedures. In particular, Admin and Operational events may be sent to their publisher, while Analytic and Debug events are immediately stored to a file because of the large volume of these types of events [21]. Due to the lack of source code of Windows Event Log framework in Microsoft Windows platform, we cannot provide its detailed architecture and implementation [21].

2.4.3 How EVTX Events work

Note that the changes in scalability and architecture that are contained in EVTX and the changes in the format itself are of equal importance [21]. Log Subscriptions is used to gather logged data from multiple hosts with EVTX enabled across the whole network [21].

2.4.3.1 Configuring Log Subscriptions

The Windows Event Collector service need to be enabled on hosts so that these hosts can collect logging data (subcriber) [21]. The Windows Remote Management service need to be enabled on any forwarding hosts (forwarder) and subscriber after configured [21]. Although

Windows supports various configuration options, the quickest configuration option is to execute "winrm quickconfig" command as a privileged user [21]. Windows Remote Management service will be setup by the command to listen on port 80/tcp [21]. Windows Firewall will be updated as well to allow this service on this port [21].

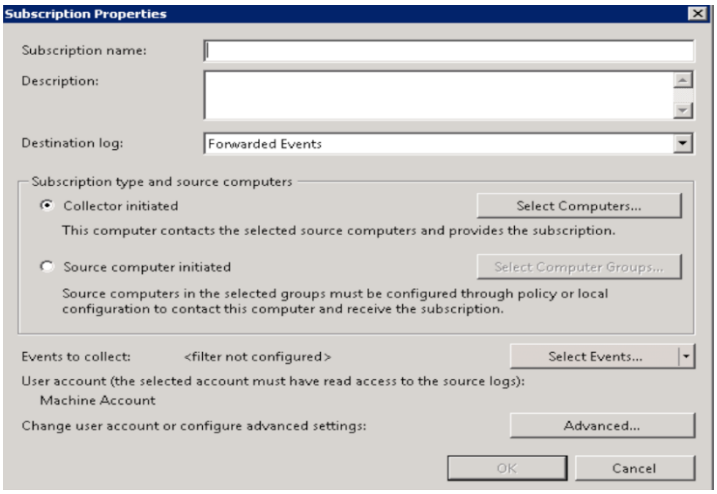

Fig. 2.10. Subscription properties wizard

Besides, appropriate permissions and/or users shall be added to the forwarding host when configuring a log subscription [21]. Windows allows Log subscriptions to be configured to use accounts on computers to forward the logging data [21].

A graphic interface inside the Subscription view is available to setup a log subscription by clicking the Create Subscription link. [21] Windows will then present the Subscription properties wizard to the user, shown in Fig 2.10.

2.4.3.2 Subscription Security

As mentioned above, subscription uses two services, Windows Event Collector service and Windows Remote Management service, to gather and forward events remotely [21]. The

main security discussion will be on Windows Remote Management service, which processes the communication between the forwarder and the subscriber [21].

Two configuration options are available to setup Windows Remote Management, which controls how to encrypt transferred data [21]. The first option involves using HTTP (TCP port 80), which transfers the data in plain text [21]. The other option will use HTTPS protocol (TCP port 43), which encrypts the data using a certificate via an SSL tunnel [21].

Windows Remote Management service provides four types of authentication that can be used to authenticate the incoming request [21]. The four types of authentication are basic, negotiate, Kerberos, and Client Certificate-based according to the Microsoft Developer Network documentation [21, 26].

Events can be forwarded from both members and non-members of a domain if configured properly.

2.5 Discussion

2.5.1 No unified data format

There are a variety of applications based on system logged data such as troubleshooting, debugging, performance tuning, auditing, IDS, digital forensic, and disaster recovery. System activities such as system calls, shared library calls in Linux, as well as event activities in Microsoft Windows can be logged. Since many regulations and standards for logging have been proposed, logging data collection is not a primary problem. With the collection problem resolved, the critical area of need is in analysis and reporting [37]. Because there is no unified

format for message content and different applications log events differently, it is difficult to analyze logged data generated from different applications, especially when these applications are from different vendors. As shown in Fig 2.11, there is a central server gathering logged data from two computers. When the logged data is collected in the two computers, it will be forwarded to the central server using syslog protocol. If the message bodies of transmitted data are in different languages, analyzing the logged data on the central server is difficult. Even when the message bodies of logged data are in the same language, the message bodies are only readable for people and are difficult for automated analyzing.

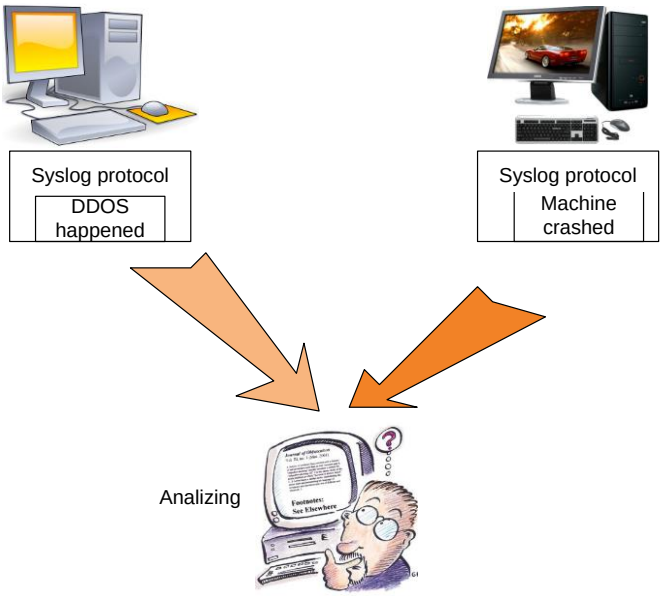

Fig. 2.11. Issues of analyzing logged data

2.5.2 Logging overhead

Since logging involves wrapping and transmitting data, it will obtain performance penalty. However, the technology that leverages the logging performance will consequently, in

turn, introduce new performance penalties. Nevertheless, there is no such research that only measures the penalty of logging itself. Much effort has been made to measure applications that involve logging functions, such as IDS and debugging tools. Limiting I/O operation to write collected data to log files has been found to reduce logging overhead [13]. Since I/O will consume many system resources, frequent I/O with a small amount of data written to files each time will dramatically impact the system. The best strategy is to write a large amount of data each time and limit I/O operation if this is tolerable. However, some applications have strict requirements embedded within them that collected data should be written to files immediately in order to empty kernel ring buffers to avoid the buffer overwrite issue. Buffer overwrite will result in data loss.

2.5.3 What level of events should be logged

Computer systems have multiple levels of events, ranging from the system start event to the kernel routine printk called event. Commonly, one high level event contains one or more lower level events. In Unix/Linux system, the lowest level event would be the execution of a single-line code. Since these trivial events occur in a high frequency in the system, a large volume of data will be logged if the application tends to log trivial events. When it comes to analyzing, more time will be consumed processing the logged data. In this case, performance penalty incurred by logging is severe. In order to reduce the volume of logged data and improve logging performance, applications could log higher level events. However, high level events such as system start and system shutdown are too abstract, and details in the system that may

interest users are lost. Therefore, the size of the logged event is critical for system performance and future processing. Currently, some applications target events at system call level.

2.5.4 Protection of logged data

As we have formerly discussed, logged data may be written as "write-once" devices to protect their integrity. The Kernel module approach and virtualization may be used as well to protect logged data. These approaches all have drawbacks. For writing logged data to "write-once" device, the compromise of associate metadata including structures and attributes of logged data may still harm the protection. For kernel module approach, suppose administrator accounts are hacked. The kernel module will be unloaded, and the whole system will be in danger. For virtualization, the performance will suffer since host computers need to virtualize all hardware resources and software environments to guest operating systems. A new approach to protect logged data is explored in the future work section.

In current operating systems, there are generally super user accounts that have unrestricted access to the operating systems [13]. In Linux, these accounts are called root accounts and in Windows they are called administrator accounts [13]. Many security models assume that these super users are trustworthy [13]. However, this assumption may be a problem for a high level security system where every user must be held accountable [13].

In order to make systems accountable, logging and auditing are used. However, logging and auditing super users' activity is circular-dependent because the super user can access and even delete the logging data [13]. True accountability (also called accountable administration) is

achieved by introducing a group of administrator accounts [13]. Illustration of the accountable administration model is shown in Fig 3.1 [13]. In the system, an administrator account maintains auditing information for all other accounts [13].

CHAPTER 3

ACCOUNTABLE ADMINISTRATION IN OPERATING SYSTEMS

3.1 Introduction

High-profile security breaches continue to emerge as computer systems gain complexity. Many efforts focus on countermeasures that detect security threats and offset security damages. Accountability complements these countermeasures. Accountability implies that an entity should be held responsible for its own specific actions [54-55, 76-77]. Once an event has transpired, it is traced back in order to determine its causes [55]. A common approach to achieve accountability is via logging and auditing [2]. Logging includes accumulating and maintaining records of system and network activities. The records are referred to as logs, logging data, audit trails, or audit logs. Auditing involves conducting reviews and examinations of system activities in order to ascertain the causes of one or more events and the responsibility of a system entity based on the logs.

In a modern operating system, there is generally a single administrative user account which has complete and unrestricted access to the system. This administrative user account is commonly referred to as the super user account. In UNIX and UNIX-like operating systems, the root is the super user account and it has all rights or permissions to all programs and files. In a Windows operating system, the Build-in Administrator also has unrestricted access and is the super user account. Since a user making use of the privilege of the super user account can make changes to any logs and erase all traces of its activity, a malicious user possessing the privilege

can launch stealth intrusions or intrusions that are difficult to account for. Many existing security models and systems are built upon the basic assumption that the super user account is trustworthy [60]. However, this assumption may not be appropriate for many mission-critical systems, for example, military systems, since a super user account can be comprised.

The common approach that attempts to achieve accountability via logging and auditing a super user's activity is a circular-dependency problem [81], in which one wants to track the super user account's activity by using logs for the super user account, but the super user account has the privilege to change and erase the logs and even the logs generated by the auditing process that analyzes the logs.

In the presence of the super user account, how does one achieve accountable system administration? In other words, how does an operating system account for all the behaviors of system administrators, in particular, the system administrators that possess the super user account and can change the logs and erase all traces of their activities? Evidently, this is a challenging problem given that many existing security frameworks are built on the assumption that the super user account is trustworthy. We argue that a truly accountable system should trust no one, not even its administrators. In this chapter, we propose a different system model in which no administrator is completely trustworthy and the concept of a single super user account is abandoned. Instead, we introduce a group of administrator accounts and then establish accountability via a logging and auditing approach. The logging component is designed such that system administrators can not alter or erase their activity log without being detected. This model

is prototyped in the Linux operating system and the performance overhead of the prototype is measured and analyzed.

Since the proposed approach suggests to create a group of peer administrator accounts and none of which has ‘super rights’, in practice a super administrator account is usually necessary in most operating systems for many cases, such as disaster recovery. This raises the question of the feasibility of the proposed approach in real world. There is a simple way to address the issue: in a scenario where proposed approach is needed, there should be a data center which does not allow remote access. If disaster recovery is needed, the administrator can use a live CD to boot the system and recover the data.

The rest of the chapter is organized as follows. Related work is presented in Section 3.2. Accoutable administration is presented in Section 3.3. We present policy and kernel primitives in Section 3.4. Implementation and evaluation are presented in Section 3.5 and Section 3.6, respectively.

3.2 Related Work

Kamp [58] created a FreeBSD tool called jail which partitions an operating system environment. In a partition (or a jail), the requests of privileged users are limited. Jail provides system administrators with the capability to delegate management capabilities for each virtual machine environment. Jail is a very useful tool for creating “virtual private machines”. However, the super user remains the omnipotent user that controls everything, including the creation and destruction of jails.

Itoi, Arbaugh, Pollack, and Reeves developed a system called sAEGIS [59] which makes use of a smart card to store computer component hashes and check against the hashes to ensure the integrity of the computer components during booting process despite the fact that system administrators are not trustworthy and may be comprised. sAEGIS enhances the system's ability to reduce the possibility of malicious software modification by administrators. However, an administrator can carry out operations that are legitimate for computer systems but considered inappropriate or even criminal in human society. For example, money laundering often involves transferring money among banks and bank accounts. The money transfers are legitimate functionality in banking systems; however, there would be a need to accounted for the transfers, should a criminal investigation were required. However, if a user making use of privilege of the super account altered and erased the traces of his or her actions, the inappropriate actions or crimes would be difficult to account for.

On Windows' platforms, users can create a append-only file through APIs such as fopen(). However, the append-only file that intends to protect log data can be deleted by system administrators. In this case, we cannot achieve accountability against system administrators. On Unix/Linux platforms, users are allowed to set file permissions to append-only. Since system administrators can still modify or even delete the file to bypass logging mechanisms, they cannot be audited if they are untrustworthy. Recent developments on SELinux make it possible to limit administrator's privileges on Linux systems, such as preventing administrator to delete or tamper a specific file. However, SELinux is implemented as a security module that can be loaded or unloaded by system administrators. In the case that a malicious system administrator unloads

SELinux at boot time, it is impossible to audit administrators through SELinux. Therefore, a protection mechanism against malicious administrators should be hardcoded to Linux kernel that cannot be bypassed by all means.

Many existing security frameworks also assume that system administrators are trusted. For example, extensive research has been dedicated to Trusted Computing. It is a basic assumption that trusted software cannot be manipulated after it is booted. Sadeghi and Stüble [60] argue that today's operating systems are definitely inappropriate for use on a trusted software basis. A key conceptual flaw is that administrators in today's operating systems can essentially bypass every security enforcement mechanism.

In summary, most security models assume that the system administrator is trustworthy. It has been witnessed [59] that the majority of security breaches come from inside of an organization and that most malicious software modifications are actually caused by administrators, comprised or not. Therefore, it is important to hold system administrators accountable.

A system must record data (e.g., audit trail or log) that represent system activities in order to achieve a certain level of accountability. Both "the Orange Book" [9] and the Common Criteria in the ISO standard 15408 [61, 10] define the requirements for auditing security related events and maintaining audit trails or logs. Windows NT 4.0 and its successors have event logs [62, 63, 64] which record security events, performance logs, and alerts. Linux [39, 65] and Sun Solaris [66] auditing systems record security-related system events.

Many researchers considered the protection of log data when the data which are intended for auditing purposes. Two aspects have been studied: 1) the operations on the audit log itself need to be audited [56], and 2) audit log data are protected and any manipulation can be either detected or prohibited [67-72]. The latter is usually achieved by using either cryptography mechanisms [72-73, 69, 71-72] or a trusted co-processor for logging [70].

It is evident that a great deal of literature dedicated to logging and auditing exist. However, the logging and auditing approach rarely is used to achieve accountable system administration.

3.3 Accountable Administration

In the proposed approach, we do not set up a single super user account. Instead, we set up multiple administrator accounts. Furthermore, there are many regular user accounts. The number of administrator accounts is denoted as N. The number N can be used to achieve different degrees of accountability. In general, the greater N, the more accountable but the more complex and the more administratively costly a system is. None of the administrator accounts have completely unrestricted power. An administrator account (denoted as A shown in Fig. 1) not only has a higher level of control than all regular user accounts, but it also maintains audit logs for the other k (k=1,…, N-1) administrator accounts. In such a case, administrator accounts are called peer administrator accounts, so that whenever any of the k peer administrator accounts does something, the event is saved in a log owned by A, which cannot be changed by the other k peer administrator accounts, as shown in Fig. 3.1, in which administrator account A audits

administrator accounts B and C as well as the regular user accounts. In the case that A finds problems with other users during an audit, a security action may be taken. Consider the following examples: 1) in a very sensitive military system, all N administrator account holders should be replaced by others to go into investigations or monitored by other “internal affair” users; 2) in a less sensitive system, a voting mechanism can be implemented among administrators to couple with some security policies to prevent voting fraud. The rationale of this proposed approach is that it is unlikely that all N administrator account holders are malicious or comprised at the same time [55], and, if using a voting approach, it is unlikely that more than L (L>=1) peer administrator account holders blame the same user wrongly. The proposed approach solves the circular-dependency problem explained earlier despite the fact that the common logging and auditing approach are still used.

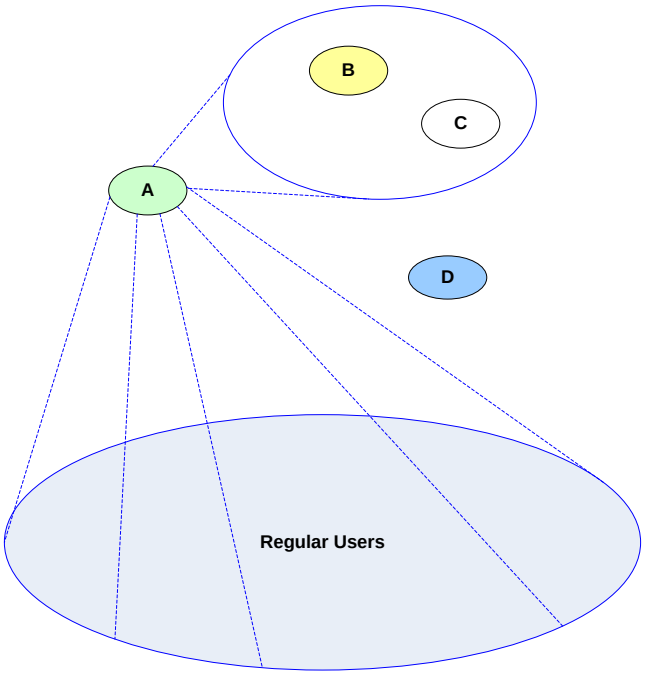

Fig. 3.1. An example of the accountable system administration

A few items are worth noting. First, a user and a user account are two different concepts. A user is a user account holder. However, an account can be held by multiple users. As an organizational policy, an organization may delegate administrator privilege to a few users. For example, in a UNIX system, many users may possess the root password and are super user account holders (or super users). However, since there is only one single super account in the operating system, these super users are indistinguishable in their activity logs. In fact, many users share a single root account makes their activity accountable more difficult. In the proposed system, each user can be assigned a peer administrator account and the activities of every peer administrator can be accountable for.

Second, the accountable system administration must be integral component of the operating system kernel and cannot be unloaded during run time. Since the accountable system administration has been hardcoded into the kernel, all system administrators can be accounted for and the logging is mandatory. However, it does not equal to that the kernel cannot be changed or upgraded. One approach can be that the system kernel may be replaced by a new version at the presence of all N peer system administrators, which will be described in a later section. Another approach can be that the system kernel must be authenticated before it replaces the current one by a single peer system administrator. Our assumption is that once the kernel is loaded, the kernel can’t be changed if without using any of the above two methods.

Third, our approach is built on logging and auditing peer system administrators’ activities in the system to hold the activities of the system administrator accounts accountable. Prevent

certain attacks to the system is not our objective. Our objective is to maintain the logging and auditing functionality no matter what happened in the system. One may argue that the logging functionality might be disabled under certain attacks. A rootkit [73] is a piece of software that enables continued privileged access to a computer, while actively hiding its presence from administrators by subverting standard operating system functionality or other applications. Typically, a hacker installs a rootkit on a computer after obtaining first non-privilege access and then privileged access, either by exploiting a known vulnerability or cracking a password. Once a rootkit is installed, it allows an attacker to mask his intrusion while gaining root or privileged access to the computer. To achieve this, a rootkit generally intercepts a number of system calls by pointing system call vectors to functions in the rootkit. Our proposed logging functionality is not based on system call interposition [75] or system call table manipulation [74]. Instead, the logging functionality is integral component of the system kernel which is not reloadable or interposable as detailed in a later section. The proposed logging functionality can not be disabled. Even these hacks may be successful; however, their behavior will be logged and can be audited at a later time.

3.4 Accountable Administration Policy and Operating System Kernel Primitives

The accountable system administration model proposed above is a general approach. In the following, we design a security policy that satisfies the requirements of such an accountable

system administration model. In addition, we provide an implementation in a Linux operating system to support this policy. The preliminary tests on our development system verified that the policy indeed satisfies the accountable system administration. Note that the proposed accountable system administration model may be satisfied by different security policies.

3.4.1 Policy of Accountable System Administration

This accountable administration security model must prevent peer administrators from altering their activity logs to be recorded by other peer administrators.

Let U and P denote the set of all users and peer administrators, respectively, and $P \subseteq U$. We call users in set $R = U - P$ regular users. An object can be a file, a folder, or a data structure. For example, a log file is such an object. Denote an object as o.

We represent the ability of user $u \in U$ to create, alter, or delete an object as a predicate $A(u, o)$. If $A(u, o) = true$, user u is capable of creating, altering, or deleting object o; otherwise, user u is not capable of creating, altering, or deleting object o. Peer administrator $p_i \in P$ monitors peer administrators $P - \{p_i\}$ and logs their activity under its own account.

The logs consist of many log objects. All log objects comprise set O. We divide system log objects O into two subsets, OP and OR, such that $O = OP \cup OR$ and $OP \cap OR = \varnothing$, where OP is the set of log objects that record peer administrators' activities and OR is the set of objects that record regular users' activity. O_{p_i} is the set of log objects which are under the account of pi

and which records other peer administrators' activities. We have $O_P = \bigcup_{p_i \in P} O_{p_i}$. The accountable system administration must satisfy the following four conditions:

$$\forall p_i, p_j \in P \text{ and } p_i \neq p_j, \ \forall o_{p_j} \in O_{p_j}, \ A(p_i, o_{p_j}) = false \quad (1)$$

$$\forall p \in P, \ \forall o_p \in O_p \ A(p, o_p) = true \quad (2)$$

$$\forall p \in P, \ \forall o_R \in O_R, \ A(p, o_R) = true \quad (3)$$

$$\forall r \in R = U - P, \ \forall o \in O = O_R \cup O_P, \ A(r, o) = false \quad (4)$$

Condition (1) maintains that peer administrators can not alter any log objects under other peer administrators' accounts. This condition ensures that a peer administrator's activity in the system is logged and can not be deleted, modified, or created. Condition (2) ensures that a peer administrator can manipulate log objects stored under its own account. As indicated in condition (1), any altering of log objects under its own account are logged under other peer administrators' accounts. Condition (3) indicates that any peer administrator can alter the objects under regular users' accounts, which implies that a peer administrator is truly more powerful than a regular user. Condition (4) is that no regular users can alter any log objects regardless of which account those objects are in.

3.4.2 Operating System Kernel Programming Primitives

To support the policy defined in the above, an operating system kernel must provide the following primitives:

1. IS_LOG(o), which returns true if o is a log object; false, otherwise;

2. GET_WHOM_LOGGED(o), which returns whose activity o represents, where o is a log object;

3. IS_ADMIN(u), which returns true if user u is a peer administrator; false if otherwise;

4. NUM_OF_ADMIN(), which returns the number of peer administrators the system currently has.

The permission of user u on log object o is then evaluated by the following predicate:

A(u, o) = (NOT IS_LOG(o)) OR (IS_ADMIN(u) AND (NUM_OF_ADMIN() == 1 OR GET_WHOM_LOGGED(o) != u))

If the above predicate evaluates to true, user u can alter o; if false, it can not. The system can define multiple sources of security policies; for example, the system can also load SELinux policies. The policies form a policy chain. The policy of accountable system administration is evaluated first in the chain of policies. When predicate A(u, o) evaluates to false, the rest of the policy chain is skipped. When an object is not a log object, predicate A(u, o) evaluates to true. The system then continues to evaluate the rest of policy chain. Whether the action is allowed depends on the rest of the policy chain.

In a system with only one administrator, A(u, o) will always evaluate to true. Thus, the system administrator's activities will not be accountable since it can remove the log objects representing its activity. Thus, more than one system administrator is required to achieve accountable system administration. In other words, when the system has only one single

administrator, our system is no different from existing system and compatible with existing systems.

Note that the policy must be loaded during system bootstrap; otherwise, the policy can be removed or unloaded and the system will not be accountable.

3.4.3 Truly Accountable System Administration

The policy and the kernel primitive guarantee a peer system administrator's activities are logged and can be audited and the peer administrator can not erase or alter the activity log. In other words, the accountability characteristic of the proposed model is provable. Since the proof is trivial and in fact already embedded in the policy and primitive design in previous subsections, we neglect the proof. Since the accountability characteristic of the proposed model is provable, the proposed model is truly an accountable system administration model. In the rest of the chapter, we describe briefly how we prototype the model and evaluate it for performance overhead.

3.5 Implementation in Linux Operating System

In the above section, we provided a policy that satisfied the requirement of the proposed accountable system administration model. The mechanism to support the policy is to let an operating system kernel provide a set of programming primitives. The support of the policy is equivalent to evaluating a predicate by using the programming primitives. In this section, we

describe our implementation in Linux operating system of such mechanism. The intention is to be as minimally invasive as possible to the system kernel, but we can demonstrate that the accountable administration is possible and beneficial to system accountability.

3.5.1 Kernel Data Structures and Primitives

3.5.1.1 Support of Peer Administrators

In a Linux system, each user is uniquely identified by its uid, or user identifier. Linux currently supports only one super user account or the root with uid 0. To minimize revision on the kernel, we denote a power user as a pair (uid, puid), where puid (standing for peer administrator identifier) is the secondary user identifier and is only meaningful when uid = 0.

3.5.1.2 Process Management

A Linux operating system is based on the process model. Any action on a log object is conducted according to a process. In Linux, the process control block that holds the control and state information of a process are represented as a task structure. The task structure is defined as struct task_struct in include/linux/sched.h where two credential attributes, struct cred *real_cred and struct cred *cred, point to the ownership of the structure and the subjects of actions invoked by the related task, respectively. The credential structures are modified so that the pair (uid, puid) becomes the credential that runs the process. Since processes invoke actions on log objects, this revision supports the implementation of the primitive IS_ADMIN(u), where u is the user whose privilege the process has.

A process may be able to spawn a child process and set its user id as a different user. For example, the Linux utility /bin/su is a setuid program which allows a non-root process to start a shell process as a root after the user provides the correct root password. In an accountable system, a peer administrator should not be allowed to impersonate another peer administrator; otherwise, the peer administrator can obtain access which should not be granted. In the current Linux system, spawning a process from a program consists of invoking two system calls: sys_execve and sys_fork. To isolate the problem, conditions are added to these two system calls. If a process running as a peer administrator requests to create a process with another peer administrator's privilege, the request will be denied.

3.5.1.3 Access Control on System Log File Objects

Linux supports different file systems. To do so, Linux defines an abstract layer, called the Virtual File System (vfs). An actual file system must provide interfaces that conform to the vfs. A basic object in vfs is inode, which represents a file, a directory, or a symbolic link. Thus, the access control on peer system administrators' logging files can be translated to access control on inodes.

In other words, the access control of logging data gives access to control of files and directories. Then all the operations are thus reduced to dealing with inodes in Linux file systems because an inode represents a basic object in a file system.

Fig. 2 shows a snapshot of a revised inode structure in Linux. Note that integer member i_puid. If i_uid is 0, the inode belongs to a log that records the activity of a peer administrator

(i_uid, i_puid). This addition supports the implementation of GET_WHOM_LOGGED(o), where o is the inode.

As shown in Fig. 3.2, another integer member i_iflog is added to the inode structure. This addition is necessary to implement primitive IS_LOG(o), where o is the inode, since this inode belongs to a log if i_iflog & 0x01 is true .

The vfs defines a few important helper functions which provide a snapshot of its defined functionalities. Fig. 3.3 shows the interfaces of a few such functions, which are found in module include/linux/fs.h. Generally speaking, when an operation is called a vfs, for example, deleting a file, the vfs will first call the helper function vfs_unlink on the corresponding inode. In this function, we check the user right expression defined in previous section A(u, o), as to determine whether the action is allowed.

```
struct inode {
struct hlist_node   i_hash;
struct list_head    i_list;
struct list_head    i_sb_list;
struct list_head    i_dentry;
……
uid_t               i_uid;
gid_t               i_gid;
uid_t               i_puid;
…...
int                 i_iflog ;

  };
```

Fig. 3.2. A revised *struct inode* for the accountable system administration

```
extern int vfs_permission(struct nameidata *, int);
extern int vfs_create(struct inode *, struct dentry *,
int, struct nameidata *);
```

```
extern int vfs_mkdir(struct inode *, struct dentry *, int);
extern int vfs_mknod(struct inode *, struct dentry *,
int, dev_t);
extern int vfs_symlink(struct inode *, struct dentry *,
const char *, int);
extern int vfs_link(struct dentry *, struct inode *,
struct dentry *);
extern int vfs_rmdir(struct inode *, struct dentry *);
extern int vfs_unlink(struct inode *, struct dentry *);
extern int vfs_rename(struct inode *, struct dentry *,
struct inode *, struct dentry *);
```

Fig. 3.3 Helper functions defined for the vfs

3.5.2 System Calls

3.5.2.1 Log File Creation

As required in the accountable administration model, peer administrators log the activity of regular users and other peer administrators. In this implementation, the logs are stored in files. A log file must be owned by a peer system administrator. In Linux, the inode structure of the file must be set with proper puid. We introduce the new system call asasetlog, where "asa" stands for "Accountable System Administration". It performs two critical operations on an open log file: 1) it sets the i_puid in the inode structure of the file it operates on to the puid of the invoking process, and 2) it sets the i_iflog in the inode structure of the file. In other words, the file becomes a log file owned by the calling peer administrator after a successful asasetlog system call. The system must be called after a file to be used as a log file has been created, generally via an open system call.

An alternative is to revise system call sys_open(), which can set the same attributes, i_puid and i_iflog, given a proper flag as an argument. In our prototype, only asasetlog() is implemented and is sufficient for evaluation purpose.

3.5.2.2 Access Control

A number of system calls must be revised to check whether the requested access permission can be granted. These system calls are related to manipulating files, and include unlink, rename, symlink, and write. Generally, the revision adds simple logic to invoke the A(u, o) primitive.

It is worth nothing that since no modification should be performed on the log file, we only allow append operating on log files. To achieve it, we make log files append-only so that peer administrator can only add entries to the log files and can not modify any entries which are already written to the log files. Our analysis indicates that the system call sys_write is critical. The method uses the revised inode data structure, in which two member variables, i_iflog and i_puid, are added to indicate whether a file is a log file and which peer administrator is the owner of the log file, respectively. The following logic is added to the sys_write system call: if a user wants to write to the log file, we first check their puid and examine whether this user is the owner of the log file. If it is the owner of the log file, we allow it to write to the log file. If it is not the owner of the log file, we only allow it to append content on the log file, as shown in Fig.3.4.

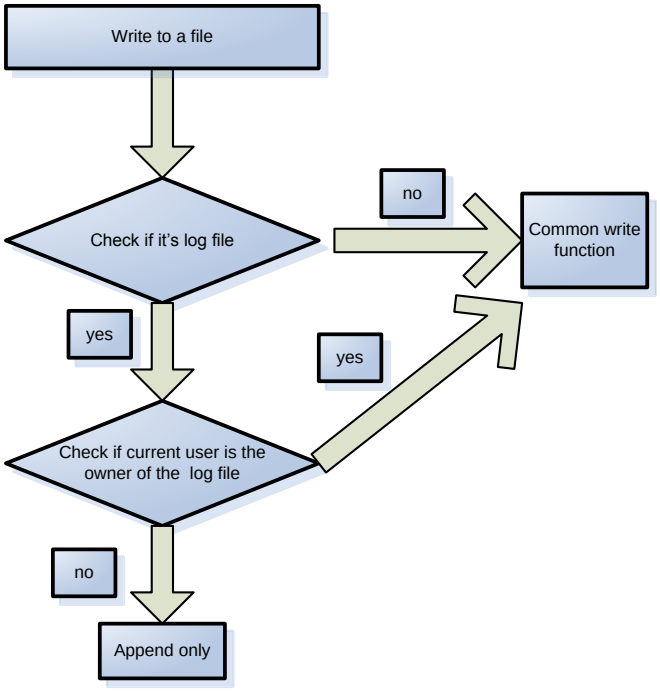

Fig. 3.4. Log files are append-only

3.5.3 Logging

The above sections guaranteed that the logging files are protected. This subsection discusses issues regarding logging, how the system records the activity of peer administrators.

The first issues to consider are the compromise of the amount of data collected and the performance overhead. On the one hand, if we log every activity of every user, the performance of the system will get worse. On the other hand, if too little data are collected regarding users' activities, we may not be able to determine the causes of one or more events. We can not achieve accountable system administration. Since regular users are prevented from altering log files due to the enforced access control mechanism, we do not need to log regular users' activities for the purpose of accountable system administration. Therefore, it is only necessary to log peer system

administrators' activities. Furthermore, log files are append-only and we only need the peer administrators' actions related to the log files. For example, when a peer administrator writes to a log file, the action is critical and is related to our security policy, so it definitely needs to be logged. The second issue to consider is that logging must be mandatory, which requires that the logging on peer system administrators' critical actions can by no means be by-passed; otherwise, the accountable system administration can not be achieved. We determine that the logging has to be performed by the kernel. The greatest events in the kernel are system calls. We identify a set of critical system calls which must be logged when they are invoked by peer system administrators. The set of systems are carefully chosen so that 1) the logging is minimal and the performance overhead is minimized and 2) the accountable system administration remains achievable. The chosen system calls are shown in Table 3.5.1.

The third issue to consider is that logging the selected system calls is equivalent to intercepting the system calls and transferring the logged events to log files in non-volatile memory. Generally, system calls in Linux can not be re-entered, though logging is performed when the system calls are being invoked. However, the logged events can not be written to logging files using the write system call; otherwise, the write system call must be re-enterable, which is challenging. Furthermore, transferring the logged events to log files immediately after the event occurs does not exploit that utilization of system resources is fluctuating in nature. It would be more beneficial for the kernel to accumulate a number of events and write the events to log files when I/O channels are idle, (i.e., scheduling can be exploited to reduce performance overhead to user processes). As outlined below, we take a two-step approach:

(1) Write the system call events to a cyclic buffer in the kernel space.

(2) A process in the user space transfers the accumulated events to log files.

In step (1), the cyclic buffer is statically allocated. The buffer size is chosen to be a few kilobytes. A kernel routine printk_log is added. The routine bears similarity to existing kernel routine printk and can be called almost anywhere and anytime in the kernel. It inserts an event record into the cyclic buffer.

In step (2), a separate process similar to syslogd is running. The process moves the events in the cyclic buffer to a given log file. When the process gets its share of CPU and moves the events to the file is at the mercy of the scheduler. Therefore, the cyclic buffer can be full before the process gets a chance to move the events. Although we use a cyclic buffer to "emulate" an infinite buffer, earlier events stored in the buffer could be overwritten by later events if proper care is not taken. In other words, we may lose some critical events. This is a typical instance of the producer-consumer problem [57]. We apply a classical solution [57] to solve the problem. The solution is to lock the cyclic buffer using the spin_lock and spin_unlock kernel routines to prevent overwritten and only allow the process to move the events to the log file. When the events are moved, some space in the cyclic buffer is made available to accommodate new events.

3.5.4 System Installation and Bootstrap

We do not allow a peer administrator to add or delete other peer administrators; in other words, we can not start with one peer administrator and add more peer administrators during the run time. We must modify the installation procedure and image (or disk) to support multiple

administrators. The system must be installed in the presence of multiple peer administrators. The accounts of the peer administrators will be created during the system installation phase.

Table 3.5.1 System calls need to be modified and logged

Name	**Synopsis**	**Descriptions**
acct	int acct(const char *filename);	switch process accounting on or off
capget, capset	int capget(cap_user_header_t header, cap_user_data_t data); int capset(cap_user_header_t header, const cap_user_data_t data);	set/get process capabilities
chmod, fchmod	int chmod(const char *path, mode_t mode); int fchmod(int fildes, mode_t mode);	change permissions of a file

chown, fchownlchown	int chown(const char *path, uid_t owner, gid_t group);
	int fchown(int fd, uid_t owner, gid_t group);
	int lchown(const char *path, uid_t owner, gid_t group);
		change ownership of a file

acct	int acct(const char *filename);	switch process accounting on or off
capget, capset	int capget(cap_user_header_t header, cap_user_data_t data); int capset(cap_user_header_t header, const cap_user_data_t data);	set/get process capabilities
chmod, fchmod	int chmod(const char *path, mode_t mode); int fchmod(int fildes, mode_t mode);	change permissions of a file

chown, fchownlchown	int chown(const char *path, uid_t owner, gid_t group);
	int fchown(int fd, uid_t owner, gid_t group);
	int lchown(const char *path, uid_t owner, gid_t group);
		change ownership of a file

link	int link(const char *oldpath, const char *newpath);	make a new name for a file
rename	int rename(const char *oldpath, const char *newpath);	change the name or location of a file
symlink	int symlink(const char *oldpath, const char *newpath);	make a new name for a file

syslog, klogctl	int syslog(int type, char *bufp, int len); int klogctl(int type, char *bufp, int len);	read and/or clear kernel message ring buffer; set console_loglevel
unlink	int unlink(const char *pathname);	delete a name and possibly the file it refers to
write	ssize_t write(int fd, const void *buf, size_t count);	write to a file descriptor
fork	pid_t fork(void);	create a child process
execve	int execve(const char *filename, char *const argv[],char *const envp[]);	execute a program
link	int link(const char *oldpath, const char *newpath);	make a new name for a file

The installation program by itself is a Linux operating system which creates file systems on a disk and copies necessary files to the disk. When the system reboots, the new kernel with accountable system administration will be loaded and in control. The number of peer system administrators and the locations of log files are system parameters which are defined during the installation phase and stored in a file whose location is defined in the new kernel. In the current implementation, the file is protected and read-only, even to the system kernel, after installation.

The protection of the parameter file is critical. We first need to guarantee the security of the parameter file and then guarantee the protection of the rest of the system. The name and the location of the parameter file have to be hard-coded in the kernel source code. Otherwise, it would become a circular-dependency problem. The protection of the parameter file is achieved via modifying the write, unlink, and rename system calls so that no one can write to, delete, or rename the parameter file.

The installation process is outlined in Fig. 3.5.

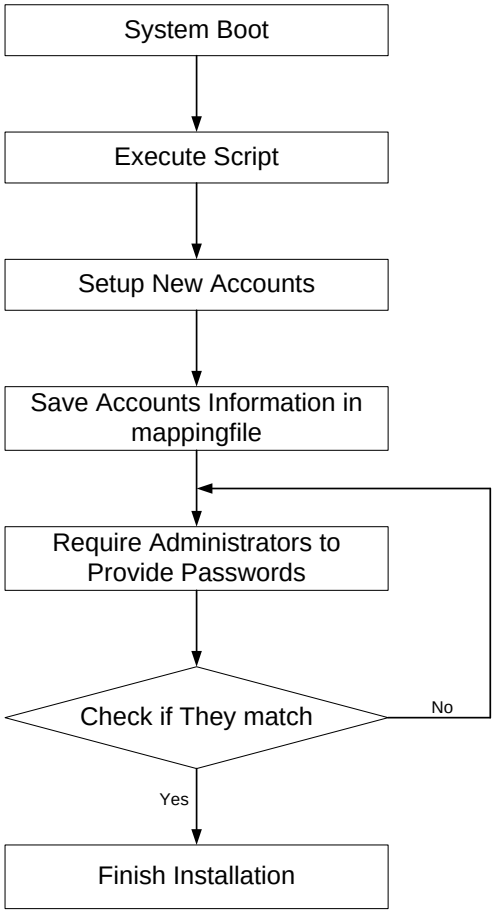

Fig. 3.5. Installation process

3.6 Evaluation

Two goals of this section are 1) to test the proposed main functionality, including the protection of the parameter file, the logging functionality, and the access control functionality and 2) to estimate the performance overhead introduced by the accountable system administration.

3.6.1 Functionality Testing

Based on Ubuntu Linux 9.10 Linux system, we created a package of revised Linux system with the proposed accountable administration. We installed the system on a computer, as described in Table 3.6.1. The protection of the parameter file, the logging functionality, and the access control functionality appear to work.

Deadlock may also happen unintentionally. To show that such a probability, if it even exists, is small, we run many programs in the testing system for an extended period of time while accumulating the logging data. No deadlock occurs when the process, known as the log mover, which moves the logging events to the log file is running.

Table 3.6.1 Implementation environment

Hardware	**Description**
system	Ubuntu 9.10
bus	0G8310
memory	512MiB DIMM SDRAM Synchronous 533*2
processor	Intel(R) Pentium(R) 4 CPU 3.00GHz
storage	82801FB/FW (ICH6/ICH6W) SATA Cont, 40GB WDC WD400BD-75JM

There is one important and subtle issue. Following the classical solution to the producer-consumer problem, we use a lock mechanism to protect the cyclic buffer, which holds logging events. When the buffer is full, the only process that is allowed to run is the log mover process. If the log mover process is not present, the system will eventually come into a deadlock, which is verified by an experiment in which the log mover process is not running. Also, this implies that the log mover process must be started earlier during the boot process, which may be regarded as a weakness of the design and implementation.

Other mechanisms exist to log system call events. For example, the Linux Audit Framework [7] uses the Netlink socket. However, those mechanisms do not consider the fact that events logged in the memory buffer may not be overwritten. The authors believe that a more radical design, such as a lock-free mechanism, may be employed to completely eliminate this weakness.

3.6.2 Performance Overhead

Operating systems are not actual user programs. The number of resources used by operating systems should be as small as possible. The accountable system administration would increase the resources used by the operating system. We would like to estimate the performance overhead introduced by the accountable system administration.

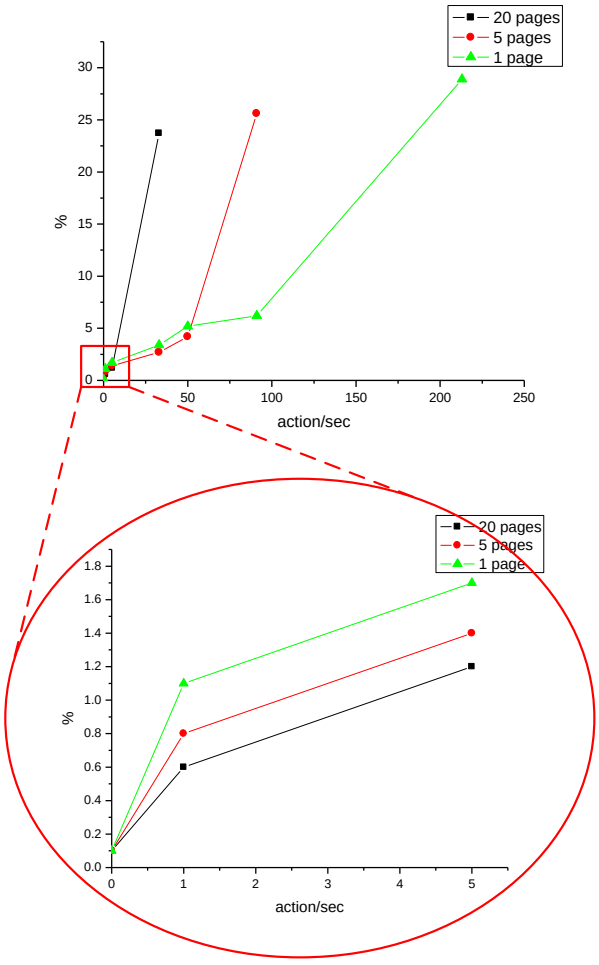

Figure 3.6 Performance overhead (accountable system administration)

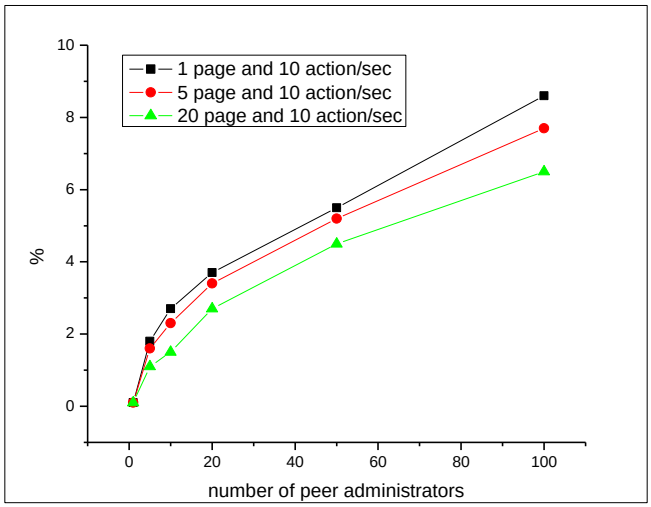

Fig. 3.7 Performance overhead (accountable system administration)

We use a worker program that opens a file, issues a number of system calls, and closes the file. The number of system calls and the frequency with which the worker program opens and closes the file has the ability to be adjusted. In other words, the worker program simulates user programs, such as web servers, file servers, and media players.

First, we run the worker program in the same computer when Ubuntu 9.10 is running and count the average running time (wall clock time). Then, we run the same worker program in the revised Linux system with the accountable system administration and record the average running time. Subsequently, we denote the former as T222off indicating that the accountable system administration is off and the later as Ton indicating that the accountable system administration is on. The performance overhead is a percentage expressed as .

There is clearly overhead because we have to add some protection logic in a few important and frequently used system calls, such as the write and delete system calls to forbid

deletion and modification actions on some protected files in addition to the logic that logs the selected system call events and move them to the log file. The performance overhead is illustrated in Figure 3.6 and Figure 3.7.

In Figure 3.6, the cyclic buffer is chosen at three different sizes: 1 page, 5 pages, and 20 pages. Since one page is 4 KB in the test system (Table 3.6.1), the buffer sizes are 4, 20, and 80 KB, respectively. The logging logic that inserts the system call events to the cyclic buffer is a CPU intensive operation, while the log mover process that moves the events from the cyclic buffer to the log file is an I/O intensive one. When the cyclic buffer becomes greater, the frequency at which the log mover actually works on its job is low, although each time it takes longer to run each time. Another dimension of the parameter is the event frequency or the number of logging events generated per second (action/second in Figure 3.6). Figure 6 show's that the overhead generally increases as the event frequency grows. However, when the event frequency is a few events per second, which should be regarded as the norm, the performance overhead (less than 2%) is considerably smaller.

In Figure 3.7, the size of cyclic buffer is chosen at three different sizes: 1 page, 5 pages, and 20 pages. The event frequency is fixed at 10 action/sec. X dimension of the parameter is the number of peer administrators. We observed that the overhead is proportional to the number of administrators. Since all the administrators will log the action of one administrator, the overhead will increase as the number of peer administrators grows.

To better explore the overhead introduced by accountable system administration, a more complicated experiment is designed. In the previous experiment, only file operations involved.

Since normal usage pattern in Unix/Linux system involves other processes with a variety of functions, we simulate these processes using another worker program. In the worker program, common Unix/Linux commands such as ls, ssh, and mkdir are included. The worker program in previous experiment will be used as well so that we can adjust the cyclic buffer as well. In Figure 3.8, we observed that the overhead introduced by accountable system administration is lower than that in previous experiment for all three sizes for a specific event frequency. Because main revision in kernel codes involves file operation system calls and a few minor revisions are in process control block hat supports multiple administrators, the overhead in Figure 3.6 can be regarded as the upper bound of overhead introduced. The more non-file operation system calls involve, the better the performance would be. The worst case would be that only file operations involved, as showed in Figure 3.6.

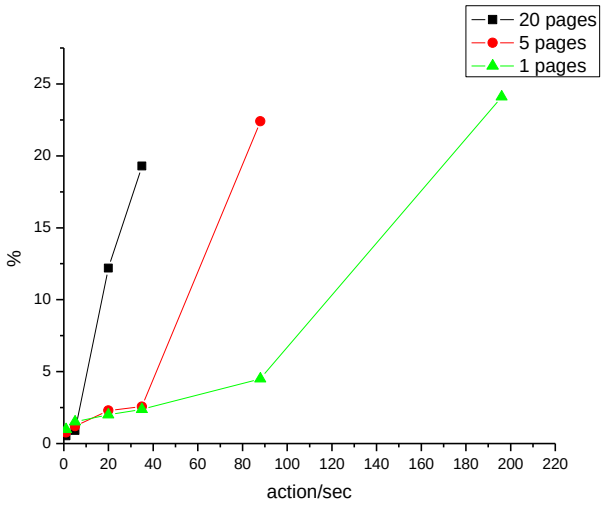

Fig. 3.8 Performance overhead (accountable system administration)

CHAPTER 4

ACCOUNTABLE LOGGING IN OPERATING SYSTEMS

4.1 Introduction

While computer system gains more complexity, security system breaches keep emerging. Many researches focus on countermeasures that detect security threats and recover the damages. Accountability contributes to these countermeasures. Accountability implies that an entity should be held responsible for its own specific actions [54, 55]. Once an event has transpired, it is traced back in order to determine its causes [55]. A common approach to achieve accountability is via logging and auditing [2]. Logging includes accumulating and maintaining records of system and network activities. Logging includes recording system activities and network activities and maintaining the recorded data at the same time. The recorded data are referred to as logs, logging data, audit trails, or even audit logs. Auditing involves conducting reviews and examinations of system activities in order to ascertain the causes of one or more events and the responsibility of a system entity based on the logs.

Syslogd and syslog-ng are the syslog daemons implemented in Linux systems. Not only can they log data from their own machines, but they may also log data from other machines [13]. Syslogd consists of two programs: klogd and syslogd. Klogd manages the logged data from the kernel, and syslogd manages the logged data from application programs, and logged data is written in log files according to the configuration files. Also, there are several applications that have the ability to produce their own logs.

In a modern operating system, there is limited logged data that is generated by the system mainly for debugging. Some security mechanisms such as SE Linux are enforced to track most of the activities in the system. However, these logging records in log files are sorted by time the event generated. Therefore, the relationships between these events are lost.

When an event triggered logging, the event information will be buffered. Then the logging module will write out the contents of the logging buffer periodically. Therefore, the timestamp of logged event is not necessarily the same as the time the event really happened. It is the time when the event is written out to the file.

When it comes to accountability, the logged events should be traced back in order to determine their causes. The relationships between these events are vital for tracing back security events. The trace back will be difficult if it only depends on the timestamp to figure out what was really happening in the system.

In this chapter, we present the design of flow-net methodology [4] and its implementation in current operating system such as Linux. The flow-net methodology not only logs the events, but also logs the relationships between the events. We also evaluate the performance for the flow-net logging implementation.

The contribution of this chapter are summarized as follows: 1) we identify the drawbacks of current logging techniques in modern operating system such as Windows and Linux; 2) a new logging technique, named flow-net is presented to eliminate identified drawbacks incurred by current logging techniques; 3) flow-net methodology is implemented in Linux; 4) the performance of implemented flow-net is evaluated.

The organization of this chapter is as follows. OS log is presented in Section 4.2. Flow-net methodology implementation in Linux is introduced in Section 4.3. Then performance evaluation is provided in Section 4.4.

4.2 Operating System Log

In this section, we will present background information for the current logging systems, such as SE Linux and Linux system logs, and analyze their generated logging files. Then we will propose our logging methodology to address the current logging system issues.

4.2.1 SELinux logging

Security-Enhanced Linux (SELinux) is a Linux feature that provides a mechanism for supporting access control security policies, including United States Department of Defense style mandatory access controls, through the use of Linux Security Modules (LSM) in the Linux kernel. It is not a Linux distribution, but rather a set of modifications that can be applied to Unix-like operating system kernels, such as Linux and that of BSD.

In Linux, there is a system log module that maintains all the buffered log events and writes out these buffers. The data generated by SE Linux is also manipulated by system log module and therefore is part of the system logs.

If the Linux Auditing System (the auditd daemon) is running, SELinux denials are logged into the audit log file. The default audit log file is /var/log/audit/audit.log. In a situation where the auditd daemon is not running, AVC denials are logged in /var/log/messages.

Once auditd is running, SELinux logs are written to the audit log file (generally /var/log/audit/audit.log). A typical snippet of an AVC denial message, apart from the time stamp in my log file, is given below and is explained in Table 4.2.1:

```
avc: denied { read } for pid=3002 comm="httpd" name="index.html" dev=hda3 ino=32004 scontext=user_u:system_r:httpd_t:s0 tcontext=system_u:object_r:tmp_t:s0 tclass=file
```

Table 4.2.1 Analysis of a SELinux logging record [15]

Message	**Description**
Avc: denied	An operation has been denied
{read}	This operation required the read permission
Pid=3002	The process with Pid 3002 executed the operation
Comm.= "httpd"	The process was an instance of the httpd program
name="index.html"	The target object was named index.html
dev=hda3	The device hosting the target object was a real disk, named hda3
ino=32004	The object was identified by the inode number 32004
scontext=user_u:system_r:httpd_t:s0	This is the security context of the process who executed the operation. It contains user, role, type and security level
tcontext=system_u:object_r:tmp_t:s0	This is the security context of the target object.
tclass=file	The target object is a file.

4.2.2 Linux system logging

Syslogd and syslog-ng are the syslog daemons implemented in Linux systems. Not only can they log data from their own machines, but they may also log data from other machines [13]. Syslogd consists of two programs: klogd and syslogd Klogd manages the logged data from the kernel, and syslogd manages the logged data from application programs, and logged data is

written in log files according to the configuration files. Also, there are several applications that have the ability to produce their own logs.

The following example is a logging record in /var/log/syslog file.

```
Feb 20 22:34:49 forrestgump-OptiPlex-755 rtkit-daemon[12467]: Successfully called chroot.
```

The first element of the logging record is timestamp, followed by a user. The application that triggers the event and the result of the event are logged as well. In this example record, process rtkit-daemon with process id 12467 successfully called chroot at 22:34:49 on Feb 20.

In essence, all the log files only contain the events with timestamps and the relationships between these events are not recorded.

4.2.3 Flow-net methodology

The flow-net methodology is illustrated in Fig. 4.1. In the figure, user A logged in, entered a directory, opened File B, read File B, closed File B, and logged out. User D logged in and created File B. The logged information includes three flows: User A, File B, and User D. The flow beginning from User D obtains three events: logging in, creating File B and logging out.

The flow-net records not only contain events information, but also contain relationships between these events. The generated data is really useful for tracing back.

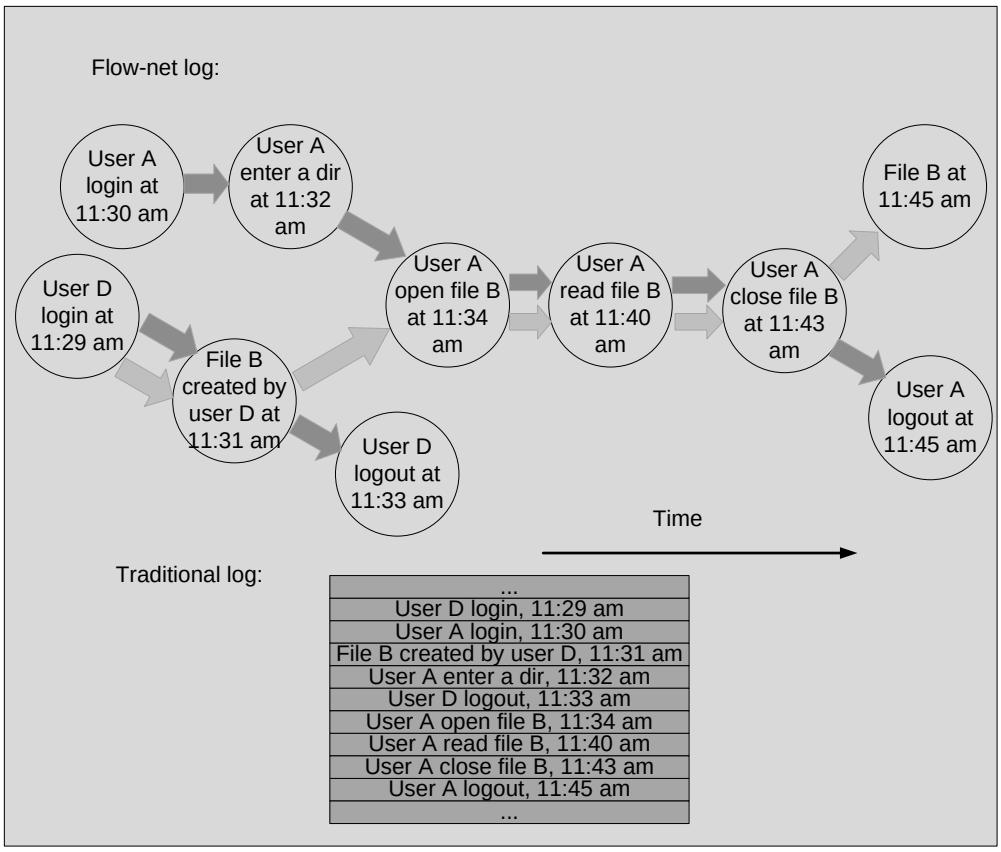

Fig. 4.1. Flow-net vs. traditional log

For traditional log, nine events in Fig 4.1 were logged with their timestamps in the generated log file. Compared to Flow-net log, the relationship between different events may only be recovered based on the timestamps and our best knowledge. However, the relationships between different flow-net events are built in the first place when the log generated.

Based on the previous analysis, we might ask: does traditional log lose any information compared with Flow-net log? When we use traditional log events to recover Flow-net log, it might be possible that the relationships between events may not be rebuilt, or may be rebuilt wrongly, based on the timestamps as well as our knowledge. We can only guess their causes and

effects when it comes to accountablilty, which is not accurate for most cases. The accuracy of these guessings are entirely based on auditor's knowledge and experience. Therefore, traditional log cannot address accountability problems.

Besides, if we can recover the relationships correctly, what is the time complexity to build Flow-net log using traditional log. Given the n recorded log events, the time complexity to build Flow-net log should be n!. At first, we need to take one event at the beginning and determine its relation with the other (n-1) events. Subsequently, take out the second event and determine its relation with the remaining (n-2) events. At last there is only one event left and rebuilding process is done.

For accountability, the logged events should be traced back in order to determine their causes. The relationships between these events are vital for tracing back security events. The trace back will be difficult if it only depends on the timestamp to figure out what was really happening in the system. Besides, traditional log only log limited security related events with the purpose of debugging, which is not suffice to answer accountability problems. For instance, some non-security related events might not be logged for the traditional log.

4.3 Flow-net Implementation in Linux

4.3.1 Common Events in Linux

If we need to log the user's activities in Linux, first we need to know the common major/large events in Linux. Therefore we make a list from the Linux system boot to power off.

- Linux boot: Every time you use a computer, you need to boot the system. Therefore Linux boot is a common event.
- Linux reboot: Reboot is also common in Linux.
- User login: When Linux boot finished, the system will require the user to log in and authenticate the users. Therefore we need to log the users who log in the system.
- Switch user: When a user logged in the system, it can switch to another user to log in the system. Because who is the current user is very important for accountability, we need to log the event.
- Open shell and close shell: Shell and any other shell version are widely used by Linux users. It is a common event to open shell and close shell.
- All shell commands that users type in: Because users can open a shell and execute many commands to do their jobs like executing a program and a killing process. The users can do anything using shell, and they also can do these things using graphic interface. Therefore we need to log everything users type to shell.
- Graphic interfaces that users use: In Linux, a user can do anything using shell and he can also do many things using graphic interfaces that application programs provide. For example, user can use a desktop shortcut to launch Firefox. He also can use "./firefox" command in shell to launch Firefox. Therefore, we also need to log these activities of using the graphic interfaces that users use. Of course we also need to log the actions of ending a program through graphic interface.

4.3.2 Event tree structure

In previous section we summarized those major events in the system. However these events are high level events and can be further divided into lower level events. In Figure 4.2 and Figure 4.3, tree structures are used to denote the two multilevel events, Linux boot and user login. Linux boot can further divided into two lower level events, PC boot and Linux process initialization. PC boot contains three lower level BIOS events, starting from BIOS checking MBR to boot loader installing operating system. Linux process initialization involves all lower level process initializations necessary for booting Linux operating system. User login event can be divided to three lower level events, initialing process to spawn getty, login process and running users' sessions. In order to spawn getty process, system needs to open tty lines, set their modes, print login prompt, get user's name and begin login process. Login process involves authenticate user name and its password. At last, users will run their own sessions.

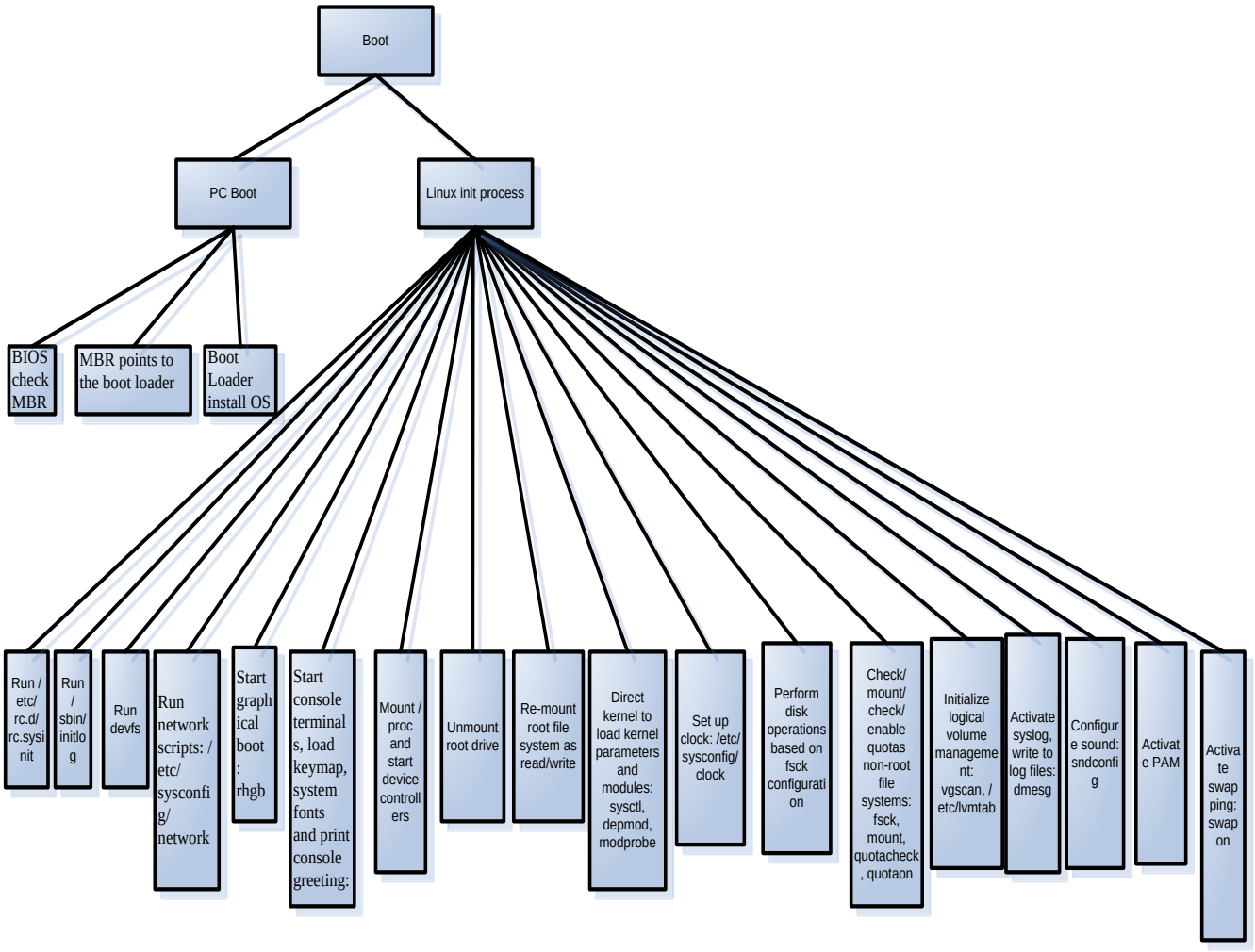

Fig. 4.2. Boot structure tree

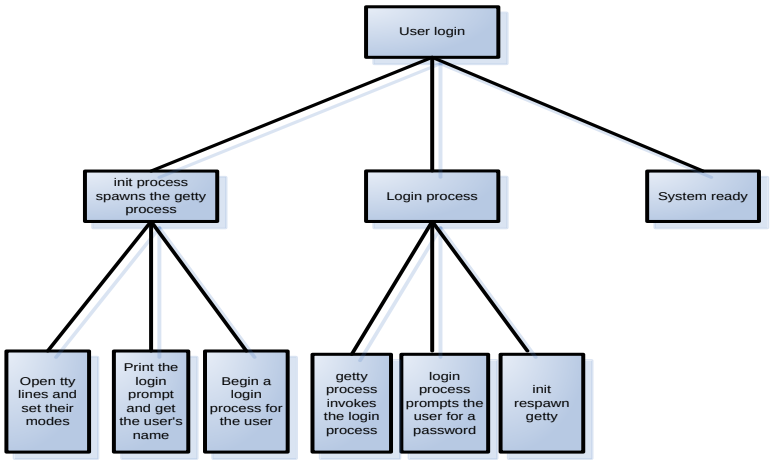

Fig. 4.3. Login structure tree

4.3.3 Logging record structure

Tokens are designed to record log information. A typical logging record consists of a sequence of logging tokens, showed below. The header toke is the beginning of a logging record. The subject token is the subject involved in the logging record and slable token holds sensitivity lable information. At last, the return token stores the status of an involved system call. .

Header token, Subject token, Slable token, Return token

There are a couple of tokens available for the accountable log system, listed in Table 4.3.1. Each token has its own format. When examining a logging record, one may see many tokens followed by data specific to these tokens. We designed a specific format for each token.

Table 4.3.2 listed format for selected tokens. For instance, “acl, user_object, user id, r” is an ACL token. It means a user has read permission on a specific object type.

Table 4.3.1 Tokens implemented in accountable log system

Message	Description
Avc: denied	An operation has been denied
{read}	This operation required the read permission
Pid=3002	The process with Pid 3002 executed the operation
Comm.= "httpd"	The process was an instance of the httpd program
name="index.html"	The target object was named index.html
dev=hda3	The device hosting the target object was a real disk, named hda3
ino=32004	The object was identified by the inode number 32004
scontext=user_u:system_r:httpd_t:s0	This is the security context of the process who executed the operation. It contains user, role, type and security level
tcontext=system_u:object_r:tmp_t:s0	This is the security context of the target object.
tclass=file	The target object is a file.

Table 4.3.2 Format of selected tokens

Token	Description
ACL	Token ID, Object type, User/group ID, permission
arbitrary	Token ID, Print format, Item size, Number items
arg	Token ID, Argument #, Argument value, Text length, text
attr	Token ID, File mode, Owner uid, File system ID, File inode id, Device id

4.3.4 Flow-net implementation in Linux

In order to capture the events occurred in the system, we want to first use the hooks added by SELinux[11] to capture the events for simplicity. Since we need to build the cross-reference structure in real time, we need to update our tree structure whenever an event occurs. Therefore we can modify the event-capturing part of SELinux and add some logic to build the cross-reference structure. Besides, we should maintain the entries to all entities in Fig 4.1. For example, user C,file A and use B are the entities in the system. In this case, we need to consider another problem: when we build a new entry for an entity? Note that all files and users are entities. If we maintain the cross-reference for all entities in the system, it will dramatically slow the system and will not have the capability of being understood. We can intuitively think that if an entity is created for the first time, we need to create a new entry for this entity. We don't need to create entries for the files created when the system is installed. When a user login the system, we need to create an entry for the user because the user may invoke many actions in the system and cause lots of logging records generated. But what if users read or write to a file created when the system is installing, which happens very often. For example, user A makes some modifications to /etc/profile to customize the system. Our solution is to create a new entry for an entity when an event involves this entity. Else we don't need to create the entry. But, if an entity involves tremendous entities, the performance of the system will be jeopardized. For example, Firefox can create many cache files and other temporary files to speed itself. When these things occur, we need to dramatically increase our cross-reference structure.

We can capture the events such as reading and writing a file in kernel. To first test whether our scheme works, we can only log the read and write event in the system and build the flow net.

In linux-2.6/fs/sysfs/bin.c:

```
Static ssize_t write(struct file *file,const char _user *userbuf,size_t bytes,loff_t *off)
Add the following codes:
#include <linux/cred.h>
#include<asm/current.h>
#include<linux/sched.h>

Static char logEventBuf[1024];
Int uid=current->real_cred->uid;
Int euid=current->real_cred->euid;
Struct dentry *dentry=file->f_path.dentry;
if  (dentry->d_inode->i_iflog==1)
    printk(KERN_INFO "%s want to write to log file %s\n",current_euid(),dentry->d_name);
logEventBuf="write";
…
```

Now we captured the write event and logged it in logEventBuf[] in kernel space. The next step is to write the data in kernel space to user space and build the cross-reference structure.

In order to build a more complicated flow, we need to capture more events. For example, login event is critical. How can we know the login event happens is also a problem. Because when the machine finishes booting, a getty process is invoked and it will invoke another process login. Therefore, a login event happens whenever a login process is invoked by exec system call. After the init process respawns the getty process, the login event has ended.

Besides all file—related operation is very important and worth logged. For example, chmod, chown and so on.

When we capture an event happening, we first check whether the involved entities are new. If these entities are new, we build a new entry for every new entity and build the link from each involved entry to the captured event. If the entities already have their entries, we traverse their event list and add the newly captured events at the end of the event list. At the same time, we have a user space program like syslog daemon to write the cross reference to a file on the disk.

During our implementation, we use an array to record these link structures.

```
struct tm {
      /*
       * the number of seconds after the minute, normally in the range
       * 0 to 59, but can be up to 60 to allow for leap seconds
       */
       int tm_sec;
       /* the number of minutes after the hour, in the range 0 to 59*/
       int tm_min;
      /* the number of hours past midnight, in the range 0 to 23 */
      int tm_hour;
       /* the day of the month, in the range 1 to 31 */
       int tm_mday;
        /* the number of months since January, in the range 0 to 11 */
        int tm_mon;
       /* the number of years since 1900 */
      long tm_year;
        /* the number of days since Sunday, in the range 0 to 6 */
      int tm_wday;
      /* the number of days since January 1, in the range 0 to 365 */
      int tm_yday;
};
```

```
Struct write {
char eventname[8]; //the name of the event
tm time;  //the time event happens
dentry file; //involved file's dentry structure
```

```
int effectiveid;  //involved user's effective id
int id;  //involved user's id
int next; //pointer to the next event
} ;
```

```
Struct read {
char eventname[8]; //the name of the event
tm time;  //the time event happens
dentry file; //involved file's dentry structure
int effectiveid;  //involved user's effective id
int id;  //involved user's id
int next; //pointer to the next event
}
```

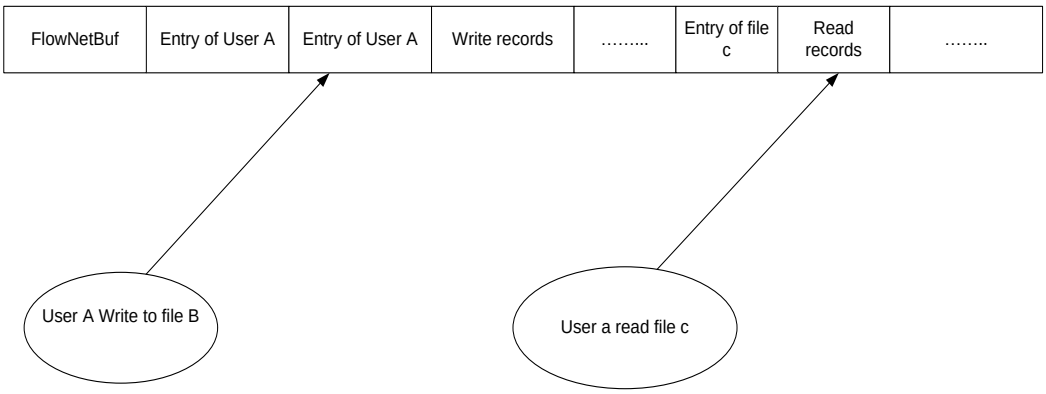

Fig. 4.4 Flow net model

4.3.5 Communication between kernel space and user space

After we have built the cross-reference structure in the array in kernel, we need a user space program to read the array out to write to the disk. There are several ways available to communicate with a user space program in kernel, such as named pipe, copy_to_user and copy_from_user and netlink socket.

In these three approaches, the named pipe is FIFO. Because we only desire to use a program in user space to write the data in array to a file, maybe there is no close relation to

FIFO. For the copy_to_user and copy_from_user approach, it will be suitable for our implementation due to thee fact that we use an array to store those cross-reference structure. copy_to_user and copy_from_user can easily copy a part of memery in kernel to user space. For the netlink approach, Netlink socket is a special IPC used for transferring information between kernel and user-space processes. It provides a full-duplex communication link between the two by way of standard socket APIs for user-space processes and a special kernel API for kernel modules [13].

In essence, we can use copy_to_user and netlink socket to write the data in kernel to user space.

Table 4.3.3 Implementation environment

Hardware	**Description**
system	Ubuntu 9.10
bus	0G8310
memory	512MiB DIMM SDRAM Synchronous 533*2
processor	Intel(R) Pentium(R) 4 CPU 3.00GHz
storage	82801FB/FW (ICH6/ICH6W) SATA Cont, 40GB WDC WD400BD-75JM

4.3.6 The benefits of Flow-net

As we all know, the minimal time unit is second/100, which is not enough to distinguish the sequence of logging event. For example, we read and write many files at the same time and it will show read and write events happened at the same time. But sometimes the sequences of read and write events are crucial for us, especially when read and write the same file. Therefore,

traditional logging may not address the problem at hand. However, it will not be a problem for the flow net because the reference itself is the actual sequence of the events.

Second, the traditional logging just show that when a file is read or wrote. But just the logging events just use a file name to distinguish different files. For instance, we open a file “file1” and don't close it we delete the “file1” and build another file named “file1” which has totally different data. Then we close the original “file1”. The logging record will be very confusing because Linux do not delete an open file until the open file is closed. Therefore, we can't figure it out what happened exactly based on the traditional logging record. If we use the flow net, the cross-references will clearly show what was going on before.

4.4 Performance Evaluation

4.4.1 Performance of Flow-net methodology implementation

When we have implemented the flow-net model in the Linux, we now desire to evaluate the system performance and check the overhead from the logging. The test scenario is that we first open a new file and write a given volume of data to the file. Subsequently, we delete the file and persist on doing the same thing for 2000 times. First we run the test program in the old default kernel and count the average running time. Next, we run the same test program in the kernel in which we implemented the flow-net model and count the average running time. Then, we may roughly estimate the overhead in the new kernel.

The program running in the old kernel took roughly 283 seconds. The program running in the new kernel took roughly 404 seconds. The performance is affected by (404-283)/283=43%.

Due to the fact that this time is for the overall running time, we cannot clearly know whether it uses the CPU cycle during the executing procedure. In Linux, we have a times () function storing current process times in struct tms:

```
struct tms {
clock_t tms_utime;  /* user time */
clock_t tms_stime;  /* system time */
clock_t tms_cutime; /* user time of children */
clock_t tms_cstime; /* system time of children */};
```

Therefore, we may use this structure to know more about the process time.

In the original system, the process took 14998(time ticks) for user time and 13426 for system time. In the new system, the process took 23695 for user time and 17254 for system time, illustrated in Table 4.4.1. Due to the logging logic in kernel, the system time is increased. In the kernel, we modified the read and write functions and everything will affect the system time. Due to the fact that the user time also increased because we launched a user space-logging program in order to write the buffer in kernel to files when we launch the test program. The two programs that run simultaneous are all IO-intensive and will wait for each other to do IO. Maybe their waiting for IO will increase the user time.

When we have implemented the accountable log model in the Linux, we then want to evaluate the system performance and check the overhead from the logging scheme. The test scenario is designed as follows: first, open a new file, write some random numbers to the file, and then delete the file. We call this set of actions “1 time action”. We record different running times for different action frequencies. We also run the test program in the old default kernel and count the average running time. Then, we run the same test program in the kernel in which we

implemented the flow-net model and count the average running time. Therefore, we may roughly get the overhead in the new kernel. The implementation environment is described in Table 4.3.3.

Table 4.4.1 System Performance of Flow-net Implementation

	User Time	**System Time**
Original System	14998	13426
System with Flow-net Implementation	23695	17254

We can see the performance result in our previous work [13]. Also, we may notice that the system performance varied by the number amount of data written to the disk. If we were to write more data to the disk in one action, the performance can be improved.

We may notice that the system performance is affected because we add some protection policy before the write () and delete () system calls to forbid delete and write actions to some protected files. We may see the performance from Fig. 5. Therefore, we can notice from the experiment that because of the SE Linux policy, the system becomes time increased. In the old kernel, we only need to run the test program. We can also see that the system performance varied by the number amount of data written to the disk. If we write more data to the disk in one action, the performance can be improved, as showed shown in Fig. 4.5.

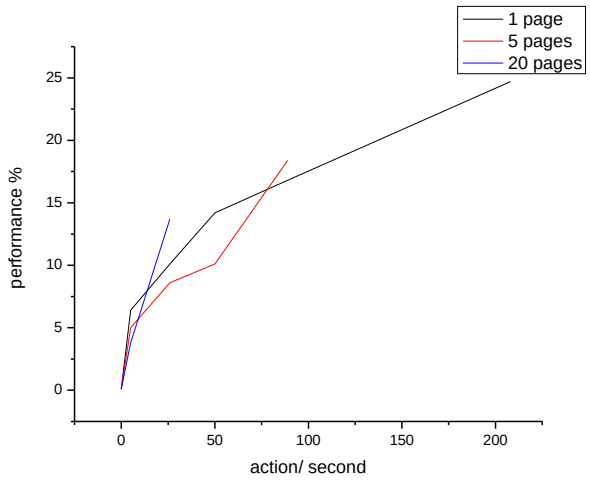

Fig. 4.5 Performance evaluation (Flow-net Model)

We may also see that the performance is improved compared to the evaluation in our previous work [13]. In that study, we used copy_to_user to deliver messages from kernel space to user space. In this study, we use netlink socket to achieve the goal. Why is the performance improved? Netlink is asynchronous because, as with any other socket API, it provides a socket queue to smooth the burst of messages. The system call for sending a netlink message queues the message to the receiver's netlink queue and then invokes the receiver's reception handler. The receiver, within the reception handler's context, may decide whether to process the message immediately or leave the message in the queue and process it later in a different context. Unlike netlink, system calls require synchronous processing. Therefore, if we use a system call to pass a message from user space to the kernel, the kernel scheduling granularity may be affected if the time to process that message is long.

4.4.2 Query Performance Using Generated Log Data

The main purpose of accountability is to trace back the events using logged data. Current data models that supported by database management system are relational database or XML database. Throughout this section, we will identify common queries in terms of accountability and evaluate the query performance using the logged data in relational database and flow-net specific database. These common queries are listed in Table 4.4.2.

Table 4.4.2 Common Queries in Terms of Accountability

	Description
Query 1	List an entity's activities in a specific time range
Query 2	List all entities' activities in a specific time range
Query 3	List an entity's activities in a specific time
Query 4	List an entity's activity

Throughout the remainder of the chapter, we will refer to these queries as Query 1, Query 2, Query 3, and Query 4, respectively. We evaluate the performance in terms of query time of traditional log and Flow-net log. For traditional log, the log records are organized by the generated time in a file, while for Flow-net log, the log records are organized as flows, as shown in Fig. 4.6. Different log records sizes are used 100, 1000,5000,10000, and 20000. The results are showed in Fig. 4.7.

Flow for entity A

Entity A	Entity B	Action	Timestamp
Entity A	Entity C	Action	Timestamp
.....			
Entity A	Entity D	Action	Timestamp

Flow for entity B

Entity B	Entity A	Action	Timestamp
Entity B	Entity E	Action	Timestamp
.....			
Entity B	Entity F	Action	Timestamp

Fig. 4.6 Logging data format (Flow-net Model)

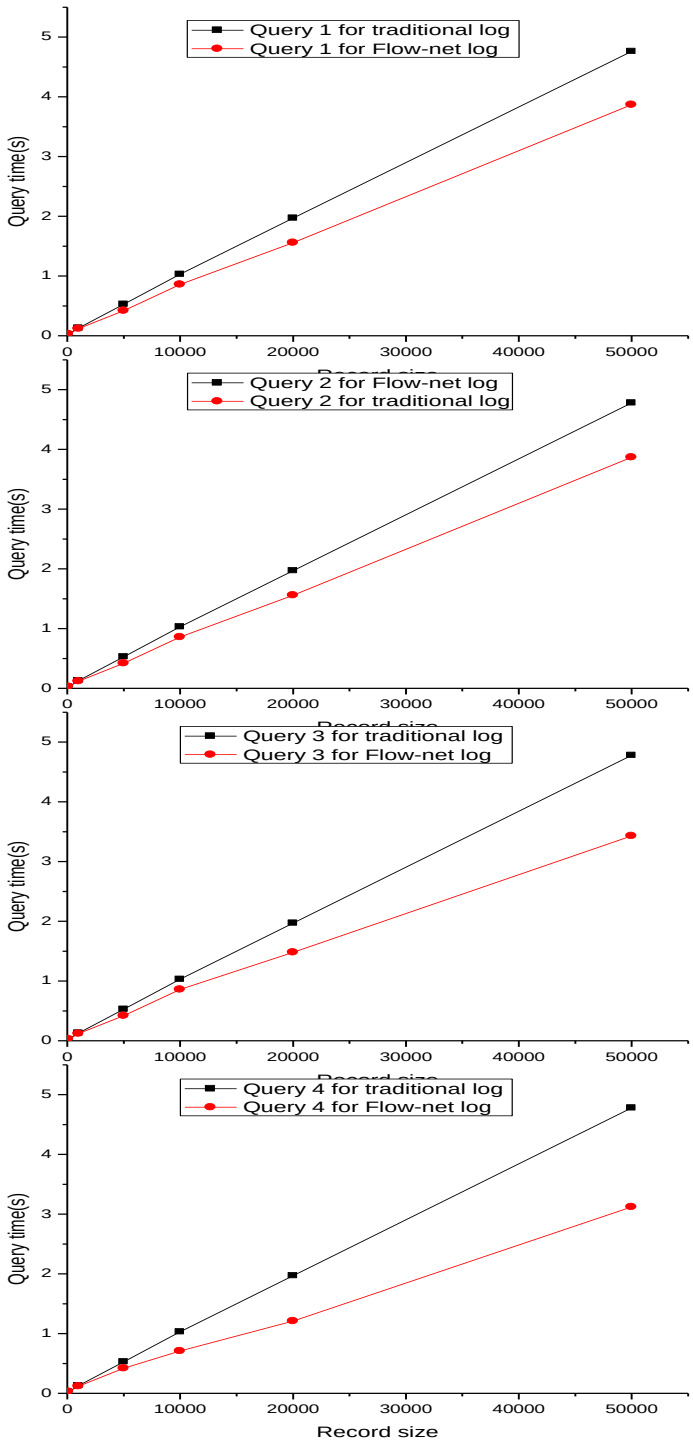
Query 1 for traditional log
Query 1 for Flow-net log
Query time(s)
0
1
2
3
4
5
0
10000
20000
30000
40000
50000
Query 2 for Flow-net log
Query 2 for traditional log
Query time(s)
Query 3 for traditional log
Query 3 for Flow-net log
Query time(s)
Query 4 for traditional log
Query 4 for Flow-net log
Query time(s)
Record size

Fig. 4.7 Query time comparison (traditional log and Flow-net log)

The query times of Query 1 for Flow-net log are shown to be less than the query time of Query 1 for traditional log. Due to the fact that to list an entity's activities in a specific time range involves checking the entire log records and print out the records that satisfy the time range requirement for traditional log. While for Flow-net log, there are references to the head of a specific flow, the log size was dramatically reduced and therefore it is in turn, less time consuming to print out the query result.

However, the query times of Query 2 for Flow-net log are shown to be greater than the query time of Query 2 for traditional log, as showed in Fig. 4.7. For Flow-net log, a specific log record is duplicated in two flows if the record involves two entities A and B (e.g., user A opened file B). Therefore, the log record exists in both flow user A and flow file B and the actual size of log records are doubled for Flow-net log structure. To list all of the entities' activities in a specific time range, it requires checking all of the log records for both Flow-net log and traditional log, and therefore the query time of Query 2 for Flow-net log will be more time consuming.

The query times of Query 3 for Flow-net log are seen to be less than the query times of Query 3 for traditional log, as showed in Fig. 4.7. Due to the fact that listing an entity's activities in a specific time involves the same operations as listing an entity's activities in a specific time range, the query times of Query 3 for Flow-net log are less time consuming.

The query times of Query 4 for Flow-net log are seen to be less than the query times of Query 4 also for traditional log use, as showed in Fig. 4.7. The operation of listing an entity's

activities involves getting to the entity's flow head and output all the records in the flow for Flow-net log, while the same operation for traditional log involves checking all the log records and output the records that satisfy the query requirement. This explained why the performance for Flow-net log is better.

CHAPTER 5

LINUX AUDITING: OVERHEAD AND ADAPTATION

5.1 Introduction

Auditing in operating systems (OSs) is necessary to achieve network and system security. Linux auditing watches file accesses, monitors system calls, records commands run by users, and security events, such as authentication, authorization, and privilege elevation, which is achieved via logging the system activities to events. Attributes of an event include the date, time, type, subject identity, result, and sensitivity labels.

The book, The Trusted Computer System Evaluation Criteria, which was published by the U.S. Department of Defense in 1985 [9] defines the requirement of logging for auditing purpose. The criteria is often referred to as "the Orange Book," which has been regarded as the security requirements for the design and implementation of secure computer systems, and has a profound influence on the design of secure computer systems. The Orange Book divides its security criteria into four categories, D, C, B, and A, which are organized in a hierarchical manner such that category D has the lowest security requirements while category A has highest ones. Category D stands for the minimal protection (i.e., the computer systems that cannot meet the requirements of the higher category). Category C sets forth the requirements for discretionary protection. Category B defines the requirements for mandatory protection. In category A, formal verification methods are required to assure that discretionary and mandatory security controls can effectively protect classified or sensitive data stored or processed by the system. Categories B and C are further divided into a number of sub classes, which are also organized in a

hierarchical manner. For example, class C1 requires the separation user and data enforcement of access limitation to the data on an individual basis while class C2 is characterized with a more finely grained, discretionary access control and thus, making users individually accountable for their actions through login procedures, auditing of security-related events, and resource isolation.

In regard to the logging functionality of computer systems, audit information must be selectively kept and protected [78]. It is the logging functionality that keeps and protects the audit information, which is often referred to as audit trails, audit data, logs, or logging data. The criteria also indicates that identifications are needed for individual subjects and access information, such as identities of accessing information, and their authorization of accessing the information is mediated [78]). This identification and authorization information must be stored securely in computers and used when performing some security-relevant actions [98]. Thus, actions affecting security can be traced to the responsible party of identity, and the outcome of the action can be assessed, which is essentially based on the logging data recorded on non-volatile memory.

The Orange Book is one of the early works that contributes to the ISO Standard 15408, "the Common Criteria for Information Technology Security Evaluation" [62], which is developed by 7 organizations, which includes the U.S. Institute of Standard and Technology (NIST) and the U.S. National Security Agent (NSA) [63]. The Common Criteria is closely related to the Orange Book. Its goal is to establish a guideline for developing and evaluating computer products in regards to its security features. The Common Criteria concentrates on the security threads from human activities. It reiterates the audit requirement for secure computer

systems. Logging is essential to meet the requirement. Although these standards recognize the importance of auditing, the focus is obviously placed in bookkeeping audit trail/log, which is in fact the system activity log.

Linux Audit Framework helps Linux meet many government and industrial security standards, such as CAPP/EAL4+ [79], LSPP [50], RBAC [80], NISPOM [81], FISMA [45], PCI [82], and DCID 6/3 [83]. It is important to the adoption of Linux in mission critical environments to meet the requirements of the security standards; otherwise, Linux cannot compete with other commercial operating systems, such as IBM, AIX, and Windows Server. They have already received certification in many government and industrial security standards.

However, to the best of our knowledge, an important question remains open: what is the performance overhead induced by Linux Audit Framework under various traffic and usage patterns, would recent development and development of high throughput/bandwidth networks further stress Linux Audit Framework?

In this chapter, we first identify the important usage patterns of Linux operating systems, and then, we design experiments to measure the overhead induced by the Linux Audit Framework in these usage patterns. The experiments inform the design of an adaptive auditing mechanism, which uses a set of selected but important type of events as the vital sign of system and network activity and uses the vital signs to adjust audit logging In order to change the type of events logged, the frequency of events logged and the time window intensive logging must be performed. The adaption achieves a low overhead in normal uses and at the same time provides

the same amount of audit data sufficiently to accomplish an audit during critical events, such as system and network intrusion.

5.2 Linux Auditing Framework

Linux audit provides users a means to analyze system activities in great detail [84]. It does not, however, protect users from attacks [84]. Instead, Linux audit helps users to track these issues and take security measures to prevent them [84].

Linux audit framework consists of several components: auditd, auditctl, audit rules, aureport, ausearch, audispd, and autrace , as shown in Fig 5.1 [84].

Auditd--The audit daemon writes the audit messages that collected in the Linux kernel to disk or transfers them to audispd [84]. The configuration file, /etc/sysconfig/auditd, controls how the audit daemon starts, and the configuration file, /etc/auditd.conf, controls how the audit daemon functions once it starts [84].

Auditctl--The auditctl utility is responsible for kernel settings, log generation parameters, and audit rules that determine which events are tracked [84].

Audit rules--Audit rules are contained in the file /etc/audit.rules. The file is loaded when the system boots and starts audit daemon [84].

Aureport--The aureport utility helps users to create a custom view of logged messages [84].

Ausearch-- The ausearch utility allows users to search the log file for certain events using the characteristics, such as keys, of the logged format [84].

Audispd--The audit dispatcher daemon (audispd) dispatches event messages to other applications instead of writing them to a disk [84].

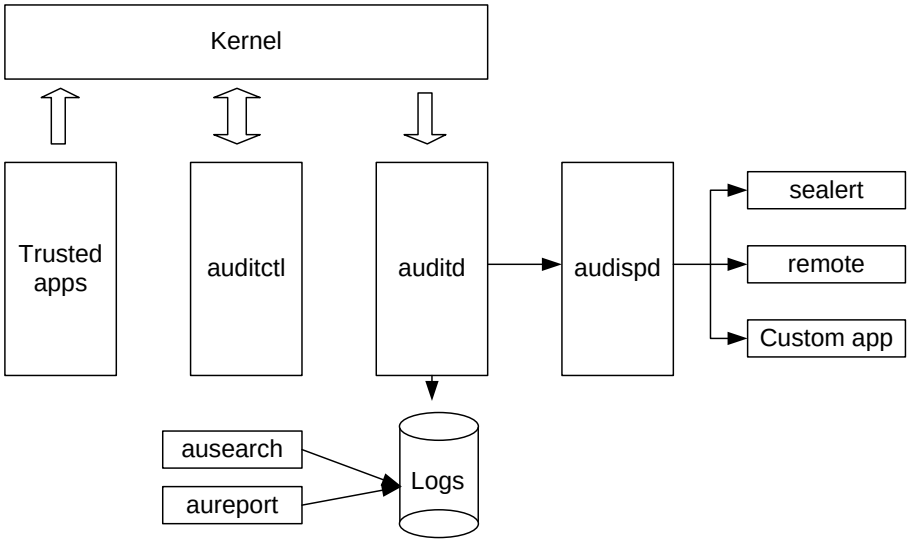

Fig 5.1. Linux auditing framework [84]

An audit event that is written to disk is shown below:

```
        type=SYSCALL msg=audit(1175176190.105:157): arch=40000003 syscall=5
success=yes
        exit=4 a0=bfba161c a1=8000 a2=0 a3=8000 items=1 ppid=4457 pid=4462 auid=0
        uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 comm="less"
        exe="/usr/bin/less" subj=unconstrained  key="LOG_audit_log"
        type=CWD msg=audit(1175176190.105:157):  cwd="/tmp"
        type=PATH msg=audit(1175176190.105:157): item=0
        name="../var/log/audit/audit.log" inode=458325 dev=03:01 mode=0100640 ouid=0
ogid=0 rdev=00:00
```

There are several keywords in an audit event record: type, msg, arch, syscall, success, exit, a0 to a3, items, ppid, pid, auid, uid, gid, euid, suid, fsuid, tty, comm., exe, subj, key, item, name, inode, dev, mode, and ouid [84]. The type field in the record denotes the type of events recorded [84]. The msg field stores a message ID [84]. The arch field refers to the CPU

architecture [84]. The syscall field stores the system call ID [84]. The success field stores the result of the system call invoked, succeeded or failed [84]. The exit field stores the return value by the system call [84]. The A0 to a3 field stores the first four arguments to the system call [84]. The items field stores the number of strings passed to the application [84]. Ppid, pid, auid, uid, gid, euid suid, fsuid, egid, sgid, and fsgid store the ID of the process, user, or group [84]. The tty field refers to the terminal from which the application is initiated [84]. Comm field stores the application name under which it appears in the task list [84]. Exe field stores the pathname of the invoked program [84]. Subj field refers to the security context to which the process is subject [84]. The key field stores the key strings assigned to each event [84]. The item field stores the path argument of system call if there is more than one argument [84]. The name field refers to the pathname passed as an argument [84]. Inode refers to the inode number corresponding to the name field [84]. The dev field stores the device and mode field stores the files access permission in numerical representation [84]. The ouid and ogid fields store the uid and gid of inode itself. Rdev is not applicable for the example record in Fig 5.2 [84].

5.3 Standard Linux Benchmarks

Table 5.3.1 lists 6 free, widely used Linux benchmark tools. These benchmark tools are open source benchmark tools.

Table 5.3.1 six free Linux benchmark tools [85]

Linux benchmark tools	**Description**
Phoronix Test Suite	Comprehensive testing
IOzone	Filesystem benchmark tool

Netperf	Network performance benchmark
LLCbench	Low level architectural benchmark
HardInfo	System profiler and benchmark
GtkPerf	GTK+ performance benchmark

5.4 Overhead of Linux Audit Logging

5.4.1 Traffic Patterns and Types of Events

Linux distributions have been used as a server operating system for its stability, security, and pricing [87]. There are several types of servers in the general network environment: web server, mail server, file server, application server, catalog server, DNS server, data server, etc [88]. Moreover, Linux is a widely used platform for cloud computing, which is a technical term that provide storage service, software, data access and computation that do not require user's knowledge of configuration of the system and physical location that provides the services [89]. Besides, Linux distributions play a major role on scientific computing because of their efficiency [90] and stability. Last but not least, Linux can be used in workstation. Current workstations uses sophisticated CPU such as Intel Xeon, AMD Opteron, or IBM Power and run Unix/Linux systems to provide reliable workhorse for computing-intensive tasks [91].

In addition, auditing is used widely in operating systems and Linux is used as a case study. More importantly, what we learned from this case study can be extended to auditing of other operating systems.

5.4.2 Experimental Design and Overhead Measurement

Similar to the logging performance study, we will run a worker program (on the host, or from the network), measure wall-clock time, with/without audit logging, and with different granularity of audit logging.

We use a worker program, which invokes a number of system calls to simulate normal workload. The number of system calls and the frequency with which the worker program opens and closes files can be adjusted. In other words, the worker program simulates normal workload in a variety of usage patterns, such as web servers, file servers, and mail servers.

First, we run the worker program on a computer with Ubuntu 9.10 running, and we count the average running time (wall clock time). Then, we run the same worker program on the same computer with auditing function enabled, and we record the average running time as well. We denote the former average running time as Toff indicating that the auditing function is off and the later as Ton indicating that the auditing function is on. The performance overhead is a percentage expressed as $C = \left(T_{on} - T_{off}\right) \Big/ T_{off}$.

5.4.3 Performance Overhead Analysis of Linux Audit Logging

Configuration of the test and evaluation computer system is shown in Table 5.4.1.

Table 5.4.1 Implementation environment

Hardware	**Description**
system	Ubuntu 9.10
bus	0G8310
memory	512MiB DIMM SDRAM Synchronous 533*2
processor	Intel(R) Pentium(R) 4 CPU 3.00GHz
storage	82801FB/FW (ICH6/ICH6W) SATA

	Cont, 40GB WDC WD400BD-75JM

Since there are 347 system calls in arch/x86/kernel/syscall_table_32.S for x86 architecture, these primary system calls in [92][93] are configured to be audited. The performance overhead is shown in Fig 5.2. In this figure, the X dimension denotes the action frequency in worker program from 1 times/s, 5 times/s, 10 times/s to 20 times/s. Because the action in worker program can only happen 23.8 times per second, the highest frequency recorded in Fig 5.2 is 20 times/s. The Y dimension denotes the overhead that computed using $C = \left(T_{on} - T_{off}\right) / T_{off}$.

Significant performance overhead is observed when action frequency is 20 times/second. With the action frequency increasing, performance overhead increases as well. Since auditing these system calls incurred system overhead, the performance penalty will increase when the frequency of invoking system calls increases.

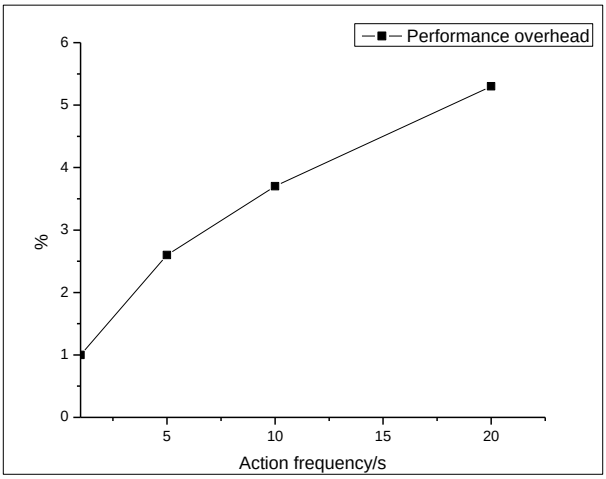

Fig 5.2. Performance overhead when auditing 347 primary system calls

5.5 Performance and Security Evaluation of Adaptive Logging

5.5.1 Performance Evaluation

This subsection shows that adaptive logging has reduced overhead.

Two goals of this section are 1) to identify critical events specific for Linux server traffic and 2) to estimate the performance overhead introduced by the auditing function.

Common tasks in the Linux server, uch as bash, aureport, crond and yum, etc., are shown in Fig 5.3. These tasks will eventually invoke the connect system call or the accept system call. In order to adapt the auditing system to the server operating system and minimize the incurred system overhead, these two critical system calls, connect and accept, are chosen to be audited.

Operating systems are not actual user programs. The number of resources used by operating systems should be as small as possible. The auditing function would increase the resources used by the operating system. We would like to estimate the performance overhead introduced by the adapted auditing function.

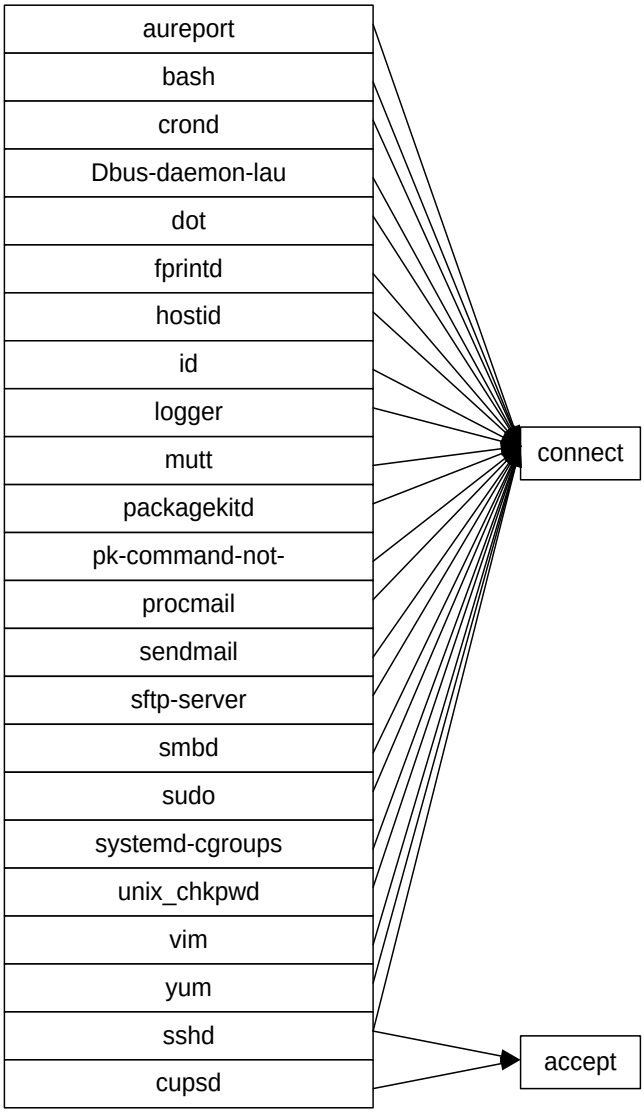

Fig 5.3. Linux tasks and corresponding system calls

Audit rules in /etc/audit/audit.rules file are shown as follows:

```
-a entry, always –S, connect
-a entry, always –S, accept
```

The first rule is to monitor the connect system call, and the second rule is to monitor the accept system call.

Figure 5.3, 5.4, 5.5, 5.6 and 5.7 are produced using aureport utility and two vital system calls, connect and accept, for web servers are monitored.

The failed access statistics report is shown in Fig 5.4. In the figure, /var/run/setrans/.setrans-unix is observed to have the highest failed access and /var/run/nscd/socket ranked second. Because /var/run/setrans/.setrans-unix contains many system related parameters that are used by a variety of applications and because only the root user hs access to this file, the other application will fail to access the file if no root privilege is obtained. The third file in Fig 5.4 is a log file owned by ssh utility and does not have a failed access recorded.

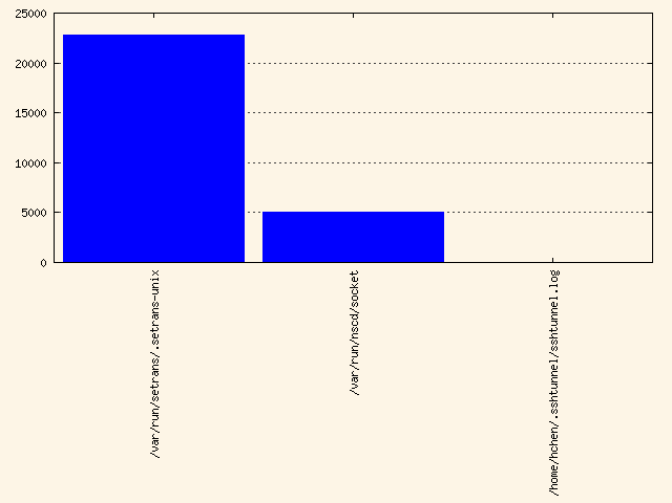

Fig. 5.4 Failed access statistics report

The system call statistics report is shown in Fig 5.5. In this figure, two system calls, connect and accept, are recorded. The connect system call was invoked many times because a lot of the common tasks of server traffic will invoke this system call.

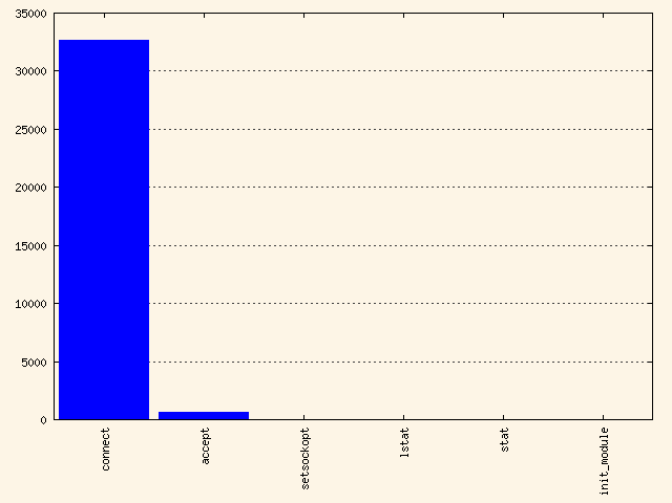

Fig. 5.5 System call statistics report

Event ranking is shown in Fig 5.6. In the figure, the system call event ranked first among all events in the server operating system.

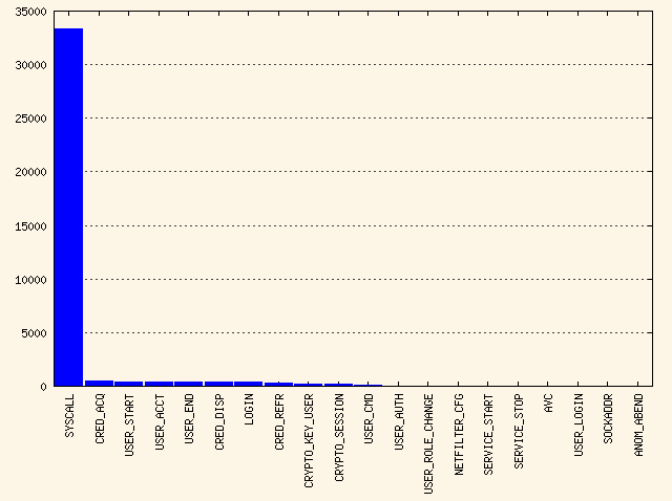

Fig 5.6 Event ranking

Non-system call event ranking is shown in Fig 5.7. The event CRED_ACQ had the highest rank among all non-system call events. For a server operating system, remote access to

the server involves obtaining credentials of different objects. The high occurrence of remote access will cause high frequency of CRED_ACQ. Furthermore, user related events, such as USER_START, USER_ACCT, and USER_END, are common as well.

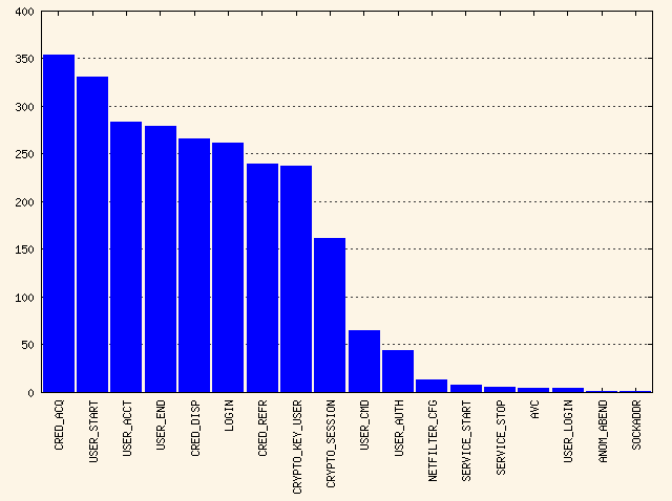

Fig 5.7 Non-system call event ranking

Performance overhead is shown in Fig 5.8. In this figure, the X dimension denotes the action frequency in the worker program from 1 times/s, 5 times/s, 10 times/s to 20 times/s. Because action in the worker program can only happen 23.8 times per second, the highest frequency recorded in Fig 8 is 20 times/s. The Y dimension denotes the overhead computed using $C = \left(T_{on} - T_{off}\right) / T_{off}$.

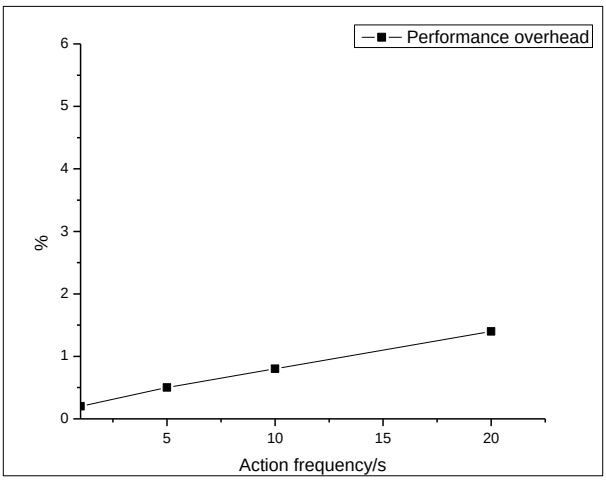

Fig. 5.8 Performance overhead

The performance overhead is observed as insignificant at a chosen frequency in the worker program, especially when compared with the previous experiment when all of the system calls are audited. This means that the performance overhead introduced by Linux auditing function is acceptable when two system calls, connect and accept, are audited. In the case that all of the system calls are audited, the performance overhead will end up being unacceptable. Therefore, auditing should be adapted to different Linux security models to enhance the security with an acceptable, incurred overhead.

5.5.2 Security Evaluation

Linux Auditing Framework helps Linux achieve a number of security standards (CAPP/EAL4+, LSPP, RBAC, NISPOM, FISMA, PCI, DCID 6/3). A brief analysis shows that the adaptive logging does not lead to any violation of the satisfied standards.

1) EAL2+

SUSE Linux Enterprise Server 8 received certification at Evaluation Assurance Level 2 (EAL2+) in 2003 [95]. This Linux distribution provided Identification and Authentication (I&A) along with basic Discretionary Access Control (DAC) capabilities [94]. Writing a Security Target using these capabilities is sufficient enough to certify a system at EAL2+ [96].

2) CAPP at EAL3+ and EAL4+

CAPP mandates auditing security relevant events and requires EAL3+ or higher [97]. Therefore, different audit systems in Linux were developed to achieve compliance with CAPP/EAL4+. Linux Audit Subsystem (LAuS) [98] and Lightweight Audit Framework (LAF) were developed by SUSE and Red Hat [99]. Although the two auditing systems share some common features, they are not compatible [94].

3) EAL4+

EAL4 provides TOE assurance by analyzing security functions [102].

4) LSPP

Labeled Security Protection Profile (LSPP) includes all the requirements from CAPP and Multi-level security [94]. Fig 5.9 showed a Multi-level Security System.

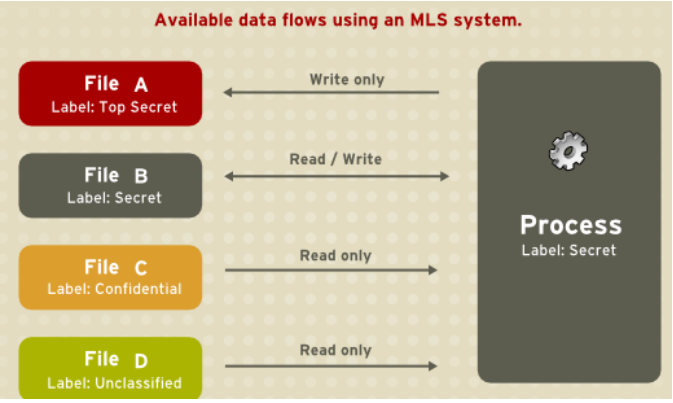

Fig 5.9 Multi-level security system [105]

5) RBACPP

RBAC model is illustrated in Fig 5.10. In this figure, there is a role that denotes Role 1, and three users that denote User4, User5, and User6. The three users have the same role and can have two types of transactions on object 1 and object 2 respectively.

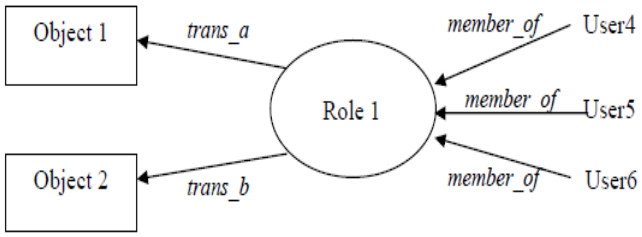

Fig 5.10 Role based access control [94]

6) NISPOM

National Industrial Security Program Operating Manual (NISPOM) require government contractors to take appropriate procedures when accessing sensitive information [100].

7) FISMA

FISMA requires government agencies to comply with certain level of security by developing programs and report when necessary [86].

8) DCID 6/3

DCID 6/3 mandates how to protect sensitive information and administrative procedures are provided [101].

9) Evaluation results

Performance overhead is shown in Fig 12. In this figure, the X dimension denotes the action frequency in the worker program from 1 times/s, 5 times/s, 10 times/s to 20 times/s. Because action in the worker program can only happen 23.8 times per second, the highest frequency recorded in Fig 12 is 20 times/s. The Y dimension denotes the overhead computed using $C = \left(T_{on} - T_{off}\right) \Big/ T_{off}$.

We can observe that RBACPP has the highest performance overhead among all for security standards, such as EAL2, EAL 4, LSPP and RBACPP. The reason is that EAL2 has minimum requirements, and RBACPP is the most restricted security standard, which involves security audit, user data protection, identification and authorization, security management, protection of target of evaluation (TOE) and TOE access. We also observed that when action frequency increases, the performance overhead also increases. Since each action, which contains 83 tasks, will trigger the Linux auditing function, more actions will cause more system overhead.

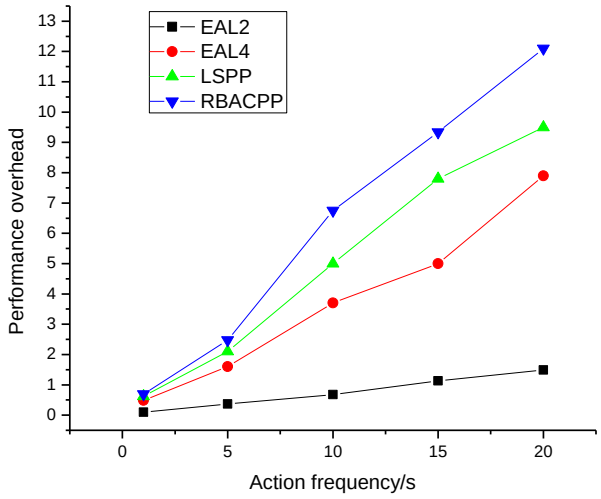

Fig 5.11 Performance overhead for different security models

5.6 Linux Benchmark Evaluation

In this section, a free Linux benchmark tool, lmbench, is used to evaluate Linux audit framework with compliance to different security standards. Lmbench is designed specifically for measuring the performance of key operations in Linux such as file system access and memory access [104].

5.6.1. Arithmetic Operation

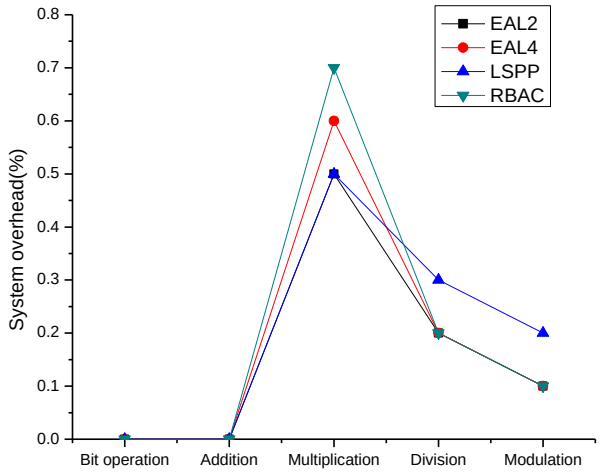

Fig 5.12 Arithmetic operation benchmarks

Figure 5.12 shows that for basic arithmetic operations, there is not much difference between these four security standards, such as EAL2, EAL4, LSPP and RBAC. The reason is that these CPU-bound tasks do not invoke any system calls that we audit.

5.6.2. Memory and File System Operations

Fig 5.13 shows no significant system overhead is observed for cache and memory access for all security standards. In the test machine, we have level 1 and level 2 caches. Since access to the memory is implemented through assembly code where no audit function is performed, we should not observe significant system overhead.

On the other hand, file system operations are considered to be sensitive operations and will introduce extensive file system related audit records. We expect to observe high system overhead. However, no significant system overhead is observed for file system operations such as file creation and file deletion.

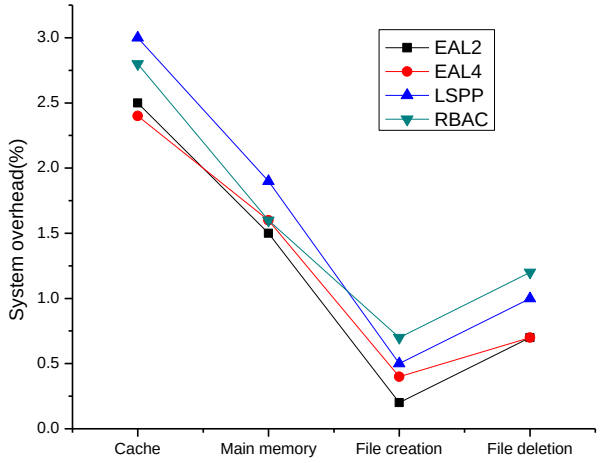

Fig 5.13 Memory and file system operations benchmarks

5.6.3. Process Related Benchmarks

Context switch is the process of storing and restoring the state of a process in order to resume execution from the same point at a later time [103]. System overhead of process related benchmarks is shown in Fig 5.14. We performed benchmarks on 8, 16 and 32 processes and associate 0, 16 and 32KB data. No significant difference for four security standards is observed for process related benchmarks.

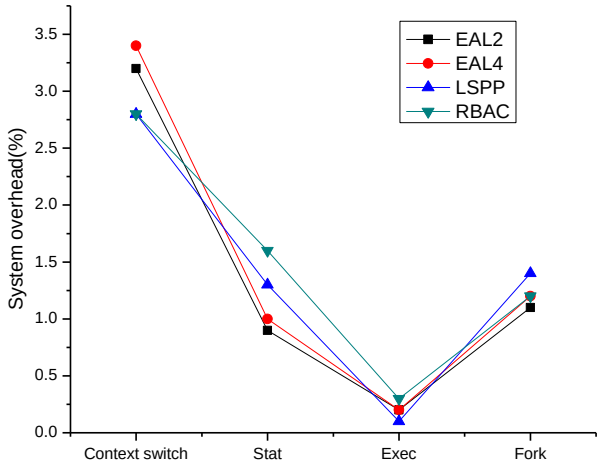

Fig 5.14 Process related benchmarks

5.6.4. Network Benchmarks

At last, we measure the performance of inter-process communications. There is not much difference, as shown in Fig 5.15, between these four security standards for different inter-process communications. RBAC security model has relatively higher system overhead over the rest three security model. The reason is role based authentication and authorization are required for RBAC and these sensitive operations will incur audit records.

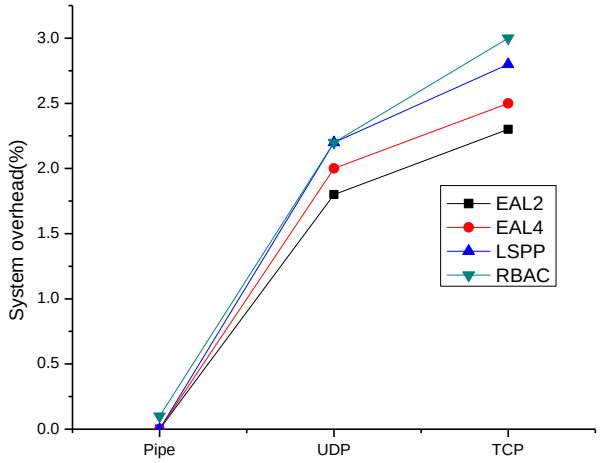

Fig 5.15 Inter-process communication for different security standards

5.6.5. Different Factors of Linux Audit Framework

Although we observed that there is a significant performance penalty when auditing is turned on on a Linux system, as shown in Fig 5.11. People may argue that the performance penalty may not be dominated by the Linux auditing alone: the factors that contribute to Linux Auditing performance penalty besides "action frequencies" that are used includes multiple sources. Imaging this, the overhead comes from many factors. A slow hard drive makes logging slow, so is a slow network. If we only store events in memory, then we remove the factors of “hard drive and network”. The factors would be I/O speed and memory speed, and concurrency control etc. Besides the factors we explored such as actions/second, file vs network, another important factor is “cyclic buffer” size. Perhaps, it can help us where the overhead come from if we vary the buffer size. The following section is to evaluate different factors of Linux audit framework.

We saved the logged data on disks, does most of penalty come from disk seek time/read/write time? If so, if we replace disks by networks (we can configure the syslogd to transport audit data to another machine), do we observe less penalty; how about we increase memory buffer.

In order to evaluate the performance penalty from disk seek time/read/write time, two test cases are designed. The first one write log data to files when necessary and the second is configured to dispatch log data to the network. Dispatching log data to network will not incur disk seek time/read/write time on test machine. Fig 5.16, the improved system performance is observed when audit daemon does not write log data to files. In conclusion, most of the penalty comes from disk seek time/read/write time.

In Fig 5.17, three test cases are designed with different buffers in kernel from 320, 640 and 1280. The cyclic buffer in the kernel space is a configuration factor in Linux Audit system: in /etc/audit/audit.rules (note the -b parameter). We observed that with bigger buffer in kernel, the curve is increasingly flatter and bigger buffer can help improve system performance in stress events. Because more buffers in kernel can improve system performance in stress events.

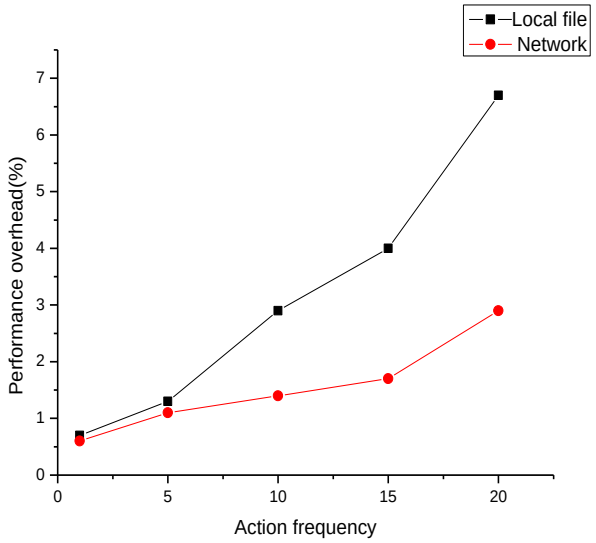

Fig 5.16 Performance overhead of audit-daemon (network)

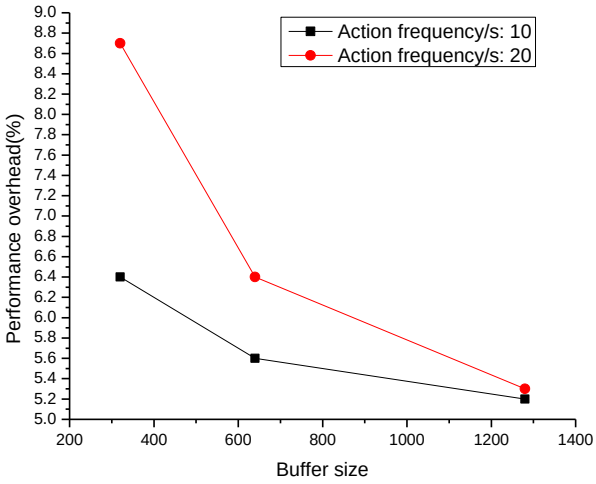

Fig 5.17 Performance overhead for different buffer sizes

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Accountable Administration in Operating Systems

In Chapter 3, we proposed a non-cryptographic approach to achieve accountable system administration. We argue that the super user is non-trustworthy and should be abandoned. We apply a common “logging” and “auditing” method for achieving accountability. The advantage of this approach is that many auditing tools and methods have already been developed and that the focus needs not be placed on “auditing”. We observed that performance is acceptable for a non-busy system with accountable administration enabled. Since the number of administrator accounts introduced will affect system overhead, we will do more testing to identify an optimal number of administrator accounts that is needed to achieve accountability while still maintain acceptable system performance.

6.2 Accountable Logging in Operating Systems

In Chapter 4, we have implemented the flow-net model to make the log accountable using the netlink approach and also written on our policy use the platform provided by SE Linux. We also evaluated the performance and made some comparison from other implementation approach. Other than that, query performance based on flow-net logging records are evaluated as well, comparing with query performance based on traditional logging records. We observed that building the Flow-net logging records will dramatically affect system performance. However,

once the Flow-net logging records are built, it will improve query time for common queries on logging records. Therefore, it is useful for systems that need to do queries repeatedly. We will do more research about how to store the flow-net format logging data in local machines. Since we might need to rebuild the flow-net structure for future queries and the action is expensive, we will try to reduce the rebuild time in the future.

6.3 Linux Auditing: Overhead and Adaptation

In Chapter 5, we introduced Linux auditing framework and measured system overhead when the auditing function was enabled. Then, we adapted the auditing function to the server traffic pattern and reevaluated system overhead. We observed that adaptive logging can dramatically reduce system overhead. Finally, we used lmbench to evaluate the performance of key operations in Linux with compliance to four security standards. We observed that local file I/O is the main factor contributing to the performance overhead for Linux Auditing framework. We also observed that the performance will be improved if the buffer size is increased. We will identify the vital signs for different usage patterns. When these vital signs are identified, we will adapt Linux auditing functions to reduce system overhead while still comply with a certain level of security standard.

REFERENCES

[1] H. Degen, “Linux in Education: Integrating a Linux Cluster into a Production High Performance Computing Environment,” vol. 2001, no.87, pp.10, 2001.

[2] Karen Kent and Murugiah Souppaya, “Guide to Computer Security Log Management, National Institute of Standards and Technology,” 2006.

[3] James Turnbull, Hardening Linux, Chapter 9 Understanding Logging and Log Monitoring, Apress, 2005.

[4] Dario V. Forte, Cristiano Maruti, Michele R.Vetturi and Michele Zambilli, SecSyslog: an Approach to Secure Logging Based on Covert Channels, Proceedings of the First International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFEf05), 2005.

[5] S. Bhansali, W. Chen, S. Jong, A. Edwards, R. Murray, M. Drinic, D.Mihocka, and J. Chau, “Framework for Instruction-Level Tracing and Analysis of Program Execution,” Proceedings of the 2nd International Conference on Virtual Execution Environments, pp. 154-163, 2006.

[6] D.L.Sallach, “A Deductive Database Audit Trail,” Proceedings of the 1922 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of 1990’s, pp. 314-319, 1990.

[7] S.T.King and P.M.Chen, “Backtracking Intrusion,” ACM Transactions on Computer Systems, vol. 23, no. 1, pp. 51-76, 2005.

[8] A.Goel, K. Farhadi, K.Po, and W. Feng, “Reconstructing System State for Intrusion Analysis,” ACM SIGOPS Operating Systems Review , vol. 42, no. 3, pp. 21-28, April 2008.

[9] US Department of Defense, “Department of Defense Trusted Computer System Evaluation Criteria,” DoD 5200.28-STD, Library No. S225,722, December 1985.

[10] The International Standard Organization, “Common Criteria for Information Technology Security Evaluation (CC),” Version 3.1, September 2006, see:http://www.commoncriteriaportal.org/public/consumer/index.php?menu=2

[11] Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos, “Log Filtering and Interpretation for Root Cause Analysis,” Software Maintenance (ICSM), 2010 IEEE International Conference.

[12] Bin-Hui Chou, Kohei Tatara, "A Secure Virtualized Logging Scheme for Digital Forensics in Comparison with Kernel Module Approach," 2008 International Conference on Information Security and Assurance.

[13] Lei Zeng, Hui Chen and Yang Xiao, "Accountable Administration and Implementation in Operating Systems," IEEE GLOBECOM 2011.

[14] D. P. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly & Associates, Sebastopol, CA, USA, 2001.

[15] R. Love. Linux Kernel Development. Novell Press, Indianapolis, IN, USA, 2nd edition, 2005.

[16] "syslog-ng—Multiplatform syslog server and logging daemon," http://www.balabit.com/network-security/syslog-ng

[17] "SANS Consensus Project Information System Audit Logging Requirements," SANS institute, 2007.

[18] Dr. Anton Chuvakin, "IT Data Management and Monitoring for ISO27000 Family of Standards," Available: http://www.loglogic.com/iso27000.

[19] "ISO27002," Available http://www.27000.org/iso-27002.htm.

[20] "The EVTX Log Format and Its Radical Impact On Existing Compliance Strategies," 2007. Doriansoftware.

[21] Brandon Charter, "EVTX and Windows Event Logging," 2008. SANS Institute.

[22] Event Properties. Retrieved July 29, 2011, from Microsoft Web site:

http://technet.microsoft.com/en-us/library/cc765981.aspx.

[23] Event Logs and Channels in Windows Event Log. from Microsoft Web site: http://msdn.microsoft.com/enus/library/aa385225.aspx.

[24] "Event Viewer-Wikipedia," available: http://en.wikipedia.org/wiki/Event_Viewer.

[25] Menn, V. (2006, November). Windows Vista: New Tools for Event Management in Windows Vista . TechNet Magazine. from Microsoft Web site: http://technet.microsoft.com/en-us/magazine/cc160886.aspx.

[26] Authentication for Remote Connections. from Microsoft Web site: http://msdn.microsoft.com/en-us/library/aa384295(VS.85).aspx.

[27] "Syslog-Wikipedia," available: http://en.wikipedia.org/wiki/Syslog.

[28] "Introduction to Syslog Protocol," available: http://www.monitorware.com/common/en/articles/syslog-described.php.

[29] "RFC 5424-The Syslog Protocol," available: http://tools.ietf.org/html/rfc5424#page-8.

[30] Raghul Gunasekaran, David A. Dillow, Galen M. Shipman, Don Maxwell, Jason J. Hill "Correlating Log Messages for System Diagnostics," Available: http://info.ornl.gov/sites/publications/files/Pub24270.pdf

[31] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and EmmettWitchel, TraceBack: first fault diagnosis by reconstruction of distributed control flow, Programming Language Design and Implementation (PLDI '05) (2005), 201–12.

[32] Nicholas Nethercote and Julian Seward, Valgrind: a program supervision framework, Electronic Notes in Theoretical Computer Science 89 (2003), no. 2.

[33] Xiangyu Zhang and Rajiv Gupta, Whole execution traces and their applications, ACM Transactions on Architecture and Code Optimization 2 (2005), no. 3, 301–334.

[34] "Intrusion detection system," http://en.wikipedia.org/wiki/Intrusion_detection_system

[35] "Debugging-Wikipedia," available: http://en.wikipedia.org/wiki/Debugging

[36] Mick Bauer, "syslog configuration," Linux Journal, Dec 2001, http://www.linuxjournal.com/article/5476

[37] Jerry Shenk,"SANS Sixth Annual Log Management Survey Report," April 2010, A SANS Whitepaper.

[38] Redhat, Inc, "Linux Auditing Discussion," available: http://www.redhat.com/mailman/listinfo/linux-audit, Sep 28, 2007.

[39] SUSE Linux AG, "Linux Audit-Subsystem Design Documentation for kernel 2.6, Version 0.1," SUSE Linux AG and Novell, Inc, 2004, available: http:///www.uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf, Sep 28,2007.

[40] US National Security Agency, "Security-Enhanced Linux," http://www.nsa.gov/selinux/, February,2008.

[41] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," Proceedings of the 11th USENIX Security Symposium, San Francisco, California, USA, August 5-9, 2002.

[42] F. Project, "SELinux Trouble Shooting Tool (setroubleshoot)," available: https://fedorahosted.org/setroubleshoot/wiki/setroubleshoot%20overview, February, 2008.

[43] "ISO/IEC 27001 - Wikipedia, the free encyclopedia," Available: ISO/IEC 27001 - Wikipedia, the free encyclopedia

[44] "Payment Card Industry (PCI) Data Security Standard—Requirements and Security Assessment Procedures version 2.0," Available: https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf

[45] "Federal Information Security Management Act of 2002-Wikipedia," Available: http://en.wikipedia.org/wiki/FISMA

[46] "Health Insurance Portability and Accountability Act - Wikipedia," Available: http://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act

[47] "COBIT-Wikipedia," Available: http://en.wikipedia.org/wiki/Cobit

[48] "Information Technology Infrastructure Library-Wikipedia," Available: http://en.wikipedia.org/wiki/Information_Technology_Infrastructure_Library

[49] "Security-evaluated operating system-Wikipedia," Available: http://en.wikipedia.org/wiki/Security-evaluated_operating_system

[50] "Labeled Security Protection Profile," October 1999, Information Systems Security Organization, Available: http://www.commoncriteriaportal.org/files/ppfiles/lspp.pdf

[51] "Role-based access control-Wikipedia," Available: http://en.wikipedia.org/wiki/RBAC

[52] "National Industrial Security Program-Wikipedia," Available: http://en.wikipedia.org/wiki/NISPOM

[53] "Sensitive Compartmented Information Facility-Wikipedia," Available: http://en.wikipedia.org/wiki/Sensitive_Compartmented_Information_Facility

[54] J. Mirkovic and P. Reiher, "Building Accountability into the Future Internet," Proceedings of the IEEE ICNP Workshop on Secure Network Protocols (NPSec), 2008.

[55] Y. Xiao, “Accountability for Wireless LANs, Ad Hoc Networks, and Wireless Mesh Networks,” IEEE Communications Magazine, Vol. 46, No. 4, Apr. 2008, pp. 116-126. 10.1109/MCOM.2008.4481350

[56] Rebecca T. Mercuri, “On auditing audit trails,” Communications of the ACM, vol. 46, no. 1, pp. 17-20, January, 2003

[57] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, Operating System Concepts, 8th Edition, Wiley, July 28, 2008

[58] P. Kamp, "Jails: Confining the omnipotent root," 2nd International System Administration and Network Engineering (SANE) Conference May 22 - 25, 2000 MECC, Maastricht, The Netherlands.

[59] N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves, "Personal Secure Booting," Lecture Notes in Computer Science, Volume 2119/2001, pp. 130-144, 2001.

[60] A. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," Proceedings of the 2004 workshop on New security paradigms, pp. 66-67, 2004.

[61] NIST ITL, “Common Criteria: Launching the International Standard,” NIST ITL Security Bulletin, National Institute of Standard and Technology, Information Technology Laboratory, November 1998, Available: http://csrc.nist.gov/publications/nistbul/11-98.pdf

[62] Microsoft, "The Threats and Countermeasures Guide," October 10, 2007, Available: http://www.microsoft.com/technet/security/guidance/serversecurity/tcg/tcgch00.mspx

[63] Microsoft, "The Audit Management in Windows 2000," October 10, 2007, Available: http://www.microsoft.com/technet/security/prodtech/windows2000/w2kccadm/auditman/w2kadm22.mspx

[64] Microsoft, "The Audit Collection Services," October 10, 2007, Available: http://technet.microsoft.com/enus/library/bb381258.aspx

[65] Redhat, Inc, “Linux Audit Discussion,” September 28, 2007 Available: http://www.redhat.com/mailman/listinfo/linux-audit

[66] Sun Microsystems, Inc, “SunSHIELD Basic Security Module Guide,” Sun Microsystems, Inc, Mountain View, CA 94303, USA, 2000, Available: http://dlc.sun.com/pdf/806-1789/806-1789.pdf

[67] Z. Baird and J. Barksdale, “Implementing a Trusted Information Sharing Environment: Using Immutable Audit Logs to Increase Security, Trust, and Accountability,” A Project of Markle Foundation, New York City, February, 2006.

[68] M. Bellare and B. Yee, “Forward Integrity for Secure Audit Logs,” Technical Report, Department of Computer Science and Engineering, University of California at San Diego, November, 1997.

[69] W. Itani, A. Kayssi, A. Chehab, and C. Gaspard, “A Policy-Driven Contact-Based Security Protocol for Protecting Audit Logs on Wireless Devices,” International Journal of Network Security, Vol. 3, No. 2, pp.124-135, September 2006.

[70] B. Schneier and J. Kelsey, “Remote Auditing of Software Outputs Using a Trusted Coprocessor,” Future Generation Computer Systems, Vol. 13, No. 1, pp.9-18, July 1997.

[71] B. Schneier and J. Kelsey, “Secure Audit Logs to Support Computer Forensics,” ACM Transactions on Information and System Security, Vol. 2, No. 2, pp.59-196, May 1999.

[72] B. Schnerer and J. Kelsey, “Cryptographic Support for Secure Logs on Untrusted Machines,” Proceedings of the 7th USENIX Security Symposium, pp.53-62, Berkeley, CA, January 1998.

[73] Z. Wang, X. Jiang, W. Cui and P. Ning, “Countering Kernel Rootkits with Lightweight Hook Protection,” Proc. of the 16th ACM Conference on Computer and Communications Security (CCS’09), pp. 545-554, 2009

[74] J. Levine, J.Grizzard, and H. Owen, “A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table,” Proc. of the 2nd IEEE International Information Assurance Workshop (IWIA’04), 2004

[75] T. Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interposition-based Security Tools,” Proc. of 2003 Network and Distributed Systems Security Symposium, 2003.

[76] Y. Xiao, "Flow-Net Methodology for Accountability in Wireless Networks," IEEE Network, Vol. 23, No. 5, Sept./Oct. 2009, pp. 30-37.

[77] Y. Xiao, K. Meng, and D. Takahashi, “Accountability using Flow-net: Design, Implementation, and Performance Evaluation," (Wiley Journal of) Security and Communication Networks, Special Issue on Security and Privacy in Emerging Information Technologies, accepted, DOI: 10.1002/sec.348.

[78] Microsoft Corporation, “How to audit user access of files, folders, and printers in Windows XP,” see: http://support.microsoft.com/kb/310399, October 10, 2007.

[79] "CAPP/EAL4+ compliant system overview," see: http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.security/doc/security/capp_compliant_overview.htm.

[80] Ravi S. Sandhu, Edward J. Coyne, "Role-Based Access Control Models," URL: http://profsandhu.com/journals/computer/i94rbac(org).pdf, 1996, IEEE.

[81] "National Industrial Security Program—Operating Manual," available: http://www.ncms-isp.org/NISPOM_200602_with_ISLs.pdf, February 2006.

[82] "Conventional PCI," available: http://en.wikipedia.org/wiki/Conventional_PCI.

[83] "Director of Central Intelligence Directive 6/3—Protecting Sensitive Compartmented Information Within Information Systems," available: http://www.fas.org/irp/offdocs/DCID_6-3_20Manual.htm.

[84] "SUSE Linux Enterprise—The Linux Audit Framework," available: http://www.suse.com/documentation/sled10/pdfdoc/audit_sp2/audit_sp2.pdf, May 8, 2008

[85] "6 best free Linux benchmark tools," available: http://www.linuxlinks.com/article/2012042806090428/BenchmarkTools.html

[86] "Federal Information Security Management Act of 2002," available: http://en.wikipedia.org/wiki/Federal_Information_Security_Management_Act_of_2002.

[87] "Wikipedia—Linux," available: http://en.wikipedia.org/wiki/Linux.

[88] "Wikipedia—Server," available: http://en.wikipedia.org/wiki/Server_(computing).

[89] "Wikipedia—Cloud Computing," available: http://en.wikipedia.org/wiki/Cloud_computing.

[90] Bill Saphir, "Linux for Scientific Computing," available: http://www.lugod.org/presentations/linux4scientificcomputing.pdf.

[91] "Wikipedia—Workstation," available: http://en.wikipedia.org/wiki/Workstation.

[92] Jialong He,"Linux System Call Quick Reference," available: http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf.

[93] "Linux System Call Reference," available: http://syscalls.kernelgrok.com/.

[94] George Wilson, Klaus Weidner, Loulwa Salem, “Extending Linux for Multi-Level Security,” available:
http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaav/SELinux/lspp-rbac.pdf.

[95] “IBM Corporation and SUSE,” IBM and SUSE earned first security certification of Linux, available: http://www-03.ibm.com/press/us/en/pressrelease/5662.wss.

[96] Atsec Gmbh and IBM Corporation, SUSE Linux Enterprise Server V8 Security Target, V1.6, 2003 available:
http://www.commoncriteriaportal.org/%20public/files/%20epfiles/0216b.pdf.

[97] ATSEC GMBH AND IBM CORPORATION. SuSE Linux Enterprise Server V 8 with Service Pack 3 Security Target for CAPP Compliance, v2.7. 2003.
http://www.commoncriteriaportal.org/public/ files/epfiles/0234b.pdf.

[98] SUSE LINUX AG. Linux Audit-Subsystem Design Documentation for Linux Kernel 2.6, v0.1, 2004. http://www.uniforum.chi.il.us/slides/ HardeningLinux/LAuS-Design.pdf.

[99] ATSEC GMBH AND IBM CORPORATION. Red Hat Enterprise Linux Version 4 Update 1 Security Target for CAPP Compliance, v2.6. 2005. http://www.commoncriteriaportal. org/public/files/epfiles/ST VID10072-ST.pdf.

[100] “Wikipedia—National Industrial Security Program,” available:
http://en.wikipedia.org/wiki/National_Industrial_Security_Program.

[101] “Protecting Sensitive Compartmented Information within Information System Manual,” available : http://www.fas.org/irp/offdocs/dcid-6-3-manual.pdf.

[102] “Evaluation assurance levels,” available:
http://cygnacom.com/labs/cc_assurance_index/CCinHTML/PART3/PART36.HTM

[103] “Context switch—Wikipedia,” available: http://en.wikipedia.org/wiki/Context_switch

[104] Qiu Jiang, “Assessing the Performance Impact of the Linux Kernel Audit Trail,” May 2004, available: http://www.scribd.com/doc/12807832/Assessing-the-Performance-Impact-of-the-Linux-Kernel-Audit-Trail

[105] “Multi-Level Security,” available:
http://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-mls-ov.html.