Linux Audit-Subsystem Design Documentation

for Kernel 2.6

Version 0.1

IBM/SUSE LINUX Confidential until Release of SLES9

Changelog

Version	Date	Authors	Reviewer	Changes, Problems, Notes
0.1	2004−03−30	Thomas Biege	Jan Beulich	− reflect new system hook design − revised ”How will events be generated?” section − revised ”What Information will be kept per Event?” section − revised ”Kernel Patch” section − erased ”Single Point of Entry...” section − revised ”Audited System Calls” section − revised ”LAuS components” − redraw some pictures

Copyright Notes

SUSE LINUX and its logo are registered trademarks of SUSE LINUX AG.
IBM and IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
Linux is a registered trademark of Linus Torvalds.
Solaris is a registered trademark of Sun Microsystems.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.
Other company, product, and service names may be trademarks or service marks of others.

The distribution and modification of this document is protected by the GNU Free Documentation Licence [4].
Copyright © 2004 SUSE LINUX AG (is a Novell company)
Copyright © 2004 Novell, Inc

Abstract

This paper describes the design of the Linux Audit Subsystem (LAuS), its components, its configuration and its CAPP compliance. LAuS was developed by Novell and SUSE LINUX to make Linux more secure and to attain the CC EAL4 certificate.

Contents

Chapter 1

Introduction

The purpose of this document is to describe the Linux Audit Subsystem (LAuS) low-level design and how it meats the requirements of Common Criteria EAL4 SUSE LINUX Enterprise Server 9.

Additionally this document serves as a communication platform for the development teams of IBM and Novell/SUSE to clarify design decisions and answer open questions.

Chapter 2

CAPP Requirements

The Controlled Access ProtectionProfile (CAPP) version 1d as released by the Information Systems Security Organization [5] lists requirements for an audit subsystem in a conforming system.

This chapter describes how the CAPP requirements are met in the LAuS implementation, along with additional requirements introduced in the Security Target (ST).

2.1 Audit Data Generation FAU_GEN.2

CAPP specifies a set of audit data generation requirements in section FAU_GEN.1.

In general, there are two mechanisms used by LAUS to generate audit data. Messages can be generated in user–space by explicitly using `laus_log()` and related library functions, and from kernel–space by intercepting system calls along with their arguments and return values, and generate audit events based on this information as specified in the audit configuration files.

The following table shows how the events required by CAPP are implemented by the LAuS system.

Explicitly generated audit messages are listed in the format "Event TYPE_subtype", system calls generated uses the format "syscall *name*". Note that some syscalls have several closely related variants, of which only the first variant is listed in the table.:

- *chmod* includes *fchmod*
- *open* includes *creat*
- *chown* includes *fchown*, *lchown*, *chown32*, *fchown32*, *lchown32*
- *setuid* includes *seteuid*, *setreuid*, *setresuid*, *setuid32*, *seteuid32*, *setreuid32*, *setresuid32*

- *setgid* includes *setegid*, *setregid*, *setresgid*, *setgid32*, *setegid32*, *setregid32*, *setresgid32*

Note that *read* and *write* system calls are not audited, because all DAC checks are done when *open*ing the file, and also because the *read*/*write* calls do not correspond directly to program actions due to buffering done by the stdio library.

CAPP Section	**Component**	**Event**	**LAuS implementation**
5.1.1	FAU_GEN.1	Start-up and shut-down of the audit functions	Events AUDIT_start, AUDIT_stop (from `auditd`)
5.1.2	FAU_GEN.2	None	
5.1.3	FAU_SAR.1	Reading of information from the audit records.	syscall *open* (on the audit log files)
5.1.4	FAU_SAR.2	Unsuccessful attempts to read information from the audit record	Like FAU_SAR.1 (syscall *open*), but with a negative result
5.1.5	FAU_SAR.3	None	
5.1.6	FAU_SEL.1	All modifications to the audit configuration that occur while the audit collection functions are operating.	Events AUDCONF_reload (generated by `auditd`); syscalls *open*, *link*, *unlink*, *rename*, *truncate* (write access to configuration files)
5.1.7	FAU_STG.2	None	
5.1.8	FAU_STG.3	Actions taken due to exceeding of threshold.	Event AUDIT_disklow (generated by `auditd`); execution of administrator-specified alert program
5.1.9	FAU_STG.4	Actions taken due to the audit storage failure	Event AUDIT_diskfull (generated by `auditd`); execution of administrator-specified alert program; all audited actions are blocked (process sleeps until space becomes available)
–	FCS_CKM.1 FCS_CKM.2 FCS_CKM.2 FCS_COP.1	None	
5.2.1	FDP_ACC.1	None	

5.2.2	FDP_ACF.1	All requests to perform an operation on an object covered by the SFP (filesystem object or IPC object).	syscalls *chmod*, *chown*, *setxattr*, *link*, *mknod*, *open*, *rename*, *truncate*, *unlink*, *rmdir*, *mount*, *umount*, *msgctl*, *msgget*, *semget*, *semctl*, *semop*, *symlink*, *removexattr*, *shmget*, *shmctl*; details include identity of object
5.2.3	FDP_RIP.2	None	
5.2.4	Note 1	None	
5.3.1	FIA_ATD.1	None	
5.3.2	FIA_SOS.1	Rejection or acceptance by the TSF of any tested secret.	Events AUTH_success, AUTH_failure (from PAM framework, "authentication" subtype)
5.3.3	FIA_UAU.1	All use of the authentication mechanism.	Events AUTH_success, AUTH_failure (from PAM framework, "authentication" subtype)
5.3.4	FIA_UAU.7	None	
5.3.5	FIA_UID.1	All use of the user identification mechanism, including the identity provided during successful attempts	Events AUTH_success, AUTH_failure (from PAM framework, subtypes "authentication" and "bad_ident"); details include origin of attempt (terminal or IP address as applicable)
5.3.6	FIA_USB.1	Success and failure of binding user security attributes to a subject (success and failure to create a process).	LOGIN audit record (from `pam_laus.so` module or `aurun`); syscalls *fork* and *clone*
5.4.1	FMT_MSA.1	All modifications of the values of security attributes of named objects (files and IPC objects).	syscalls *chmod*, *chown*, *setxattr*, *msgctl*, *semctl*, *shmctl*

5.4.2	FMT_MSA.3	Modifications of the default setting of permissive or restrictive rules. All modifications of the initial value of security attributes.	syscalls *umask*, *open*
5.4.3	FMT_MTD.1	All modifications to the values of TSF data (create, delete and clear the audit trail).	syscalls *open*, *rename*, *link*, *unlink*, *truncate* (of audit log files)
5.4.4	FMT_MTD.1	All modifications to the values of TSF data (modify or observe the set of audited events).	syscalls *open*, *link*, *rename*, *truncate*, *unlink* (of audit config files); event AUDCONF_reload.
5.4.5	FMT_MTD.1	All modifications to the values of TSF data (initialize and modify user attributes other than authentication data).	"gpasswd" audit text messages (from shadow suite), details include new value of of the TSF data
5.4.6	FMT_MTD.1	All modifications to the values of TSF data (initialize and modify user authentication data).	"gpasswd" audit text messages (from shadow suite); attempts to bypass trusted programs detected through audited syscalls *open*, *rename*, *truncate*, *unlink*
5.4.7	FMT_REV.1	Revocation of user attributes	Event: "gpasswd" audit text messages (from shadow suite); attempts to bypass trusted programs detected through audited syscalls *open*, *rename*, *truncate*, *unlink*
5.4.8	FMT_REV.1	Revocation of object attributes	system calls *chmod*, *chown*, *setxattr*, *unlink*, *truncate*, *msgctl*, *removexattr*, *semctl*, *shmctl*

5.4.9	FMT_SMR.1	Modifications to the group of users that are part of a role.	Event: “gpasswd:” audit text messages “group member added”, “group member removed”, “group administrators set”, “group members set” (from trusted programs in shadow suite).
5.4.9	FMT_SMR.1	Every use of the rights of a role (Additional/Detailed)	The user’s actions result in audited syscalls and the use of trusted programs that are audited. Details include the login ID, the origin can be determined from the associated LOGIN record for this login ID and audit session ID.
5.5.1	FPT_AMT.1	Execution of the test of the underlying machine and the result of the test.	Event: ADMIN_amtu (generated by AMTU testing tool)
5.5.2	FPT_RVM.1	None	
5.5.3	FPT_SEP.1	None	
5.5.4	FPT_STM.1	Changes to the time.	Event: syscalls *(do_)settimeofday*, *adjtimex*
–	FTP_ITC.1	Set-up of trusted channel.	Event: syscall *exec* (of `stunnel` program)

2.2 User Identity Association FAU_GEN.2

To keep track of the owner of a process and to keep an audit trail for an interactive user session a “Login ID“ is associated with every process. The “Login ID“ gets inherited if a process spawns a new process. For example, this enables the Security Officer (SO) to determine the real owner of a malicious process even if the user changes his “User IDs“.

2.3 Audit Review FAU_SAR.1

LAuS provides a user space tool, `aucat`, that translates the on-disk binary format to a human readable format at the request of an authorized administrator.

2.4 Restrict Audit Review FAU_SAR.2

The audit log file is protected by DAC controls so that only an authorized administrator is able to read the logs. The audit tools are also protected by DAC controls so that only authorized administrators can invoke the tools.

2.5 Selectable Audit Review FAU_SAR.3

LAuS provides a user space tool, `augrep`, that allows the administrator to filter the audit records to only display requested events. The administrator is able to filter on:

- user
- group
- syscall
- file
- file operations
- outcome (success/failure)
- remote hostname
- remote hostname address
- audit ID
- syscall arguments

2.6 Selective Audit FAU_SEL.1

LAuS gives the administrator the ability to select the events to audit. This is done by the administrator editing the filter configuration file of the audit daemon and then running `auditd -r` to notify the audit daemon of the change in configuration. The audit daemon in turn notifies the kernel of the new auditing policy.

2.7 Guarantees of Data Availability FAU_STG.1

LAuS prevents unauthorized deletion and modification of audit records via DAC controls.

2.8 Action in Case of Audit Data Loss FAU_STG.3

If the system runs out of disk space, the audit daemon will stop reading from the device file which will result in filling up the buffers of the audit subsystem. Subsequently, the kernel will block any process trying to enqueue new audit events for delivery to the audit daemon. To ensure that this blocking happens as soon as possible, the audit daemon sets the message queue length to zero when detecting an out-of-space condition.

2.9 Prevention of Audit Data Loss FAU_STG.4

To avoid the loss of data, multiple "bin files" are used. Each file has a fixed size. If one file is full, it will be locked and processed by external commands specified in the configuration file. During that time, the next bin file is used for storing audit records. If the external command fails (exits with a non-zero exit status), the SO will be notified through syslog and the audit system will be suspended.

2.10 Management of the Audit Trail FMT_MDT.1

The LAuS log files can be added to the set of audited objects to detect malicious modifications of the audit trail. Furthermore, only the superuser is able to access the audit trail due to the appropriate DAC settings of the file(s).

2.11 Management of audited Events FMT_MDT.1

A user can not modify the set of audit events that is generated due to his or her activity unless he is the superuser. Only the superuser is able to communicate with the kernel and to modify the configuration files of the audit daemon.

2.12 Reliable Time Stamps FPT_STM.1

LAuS uses the system time and only the superuser is able to modify the system time.

Chapter 3

High Level Design

The sections of this chapter try to clarify the abstract behavior of the Linux Audit subsystem. The sections are ordered by data flow to make it more logical to the reader to understand.

(Please note that every action to configure or modify the audit subsystem has to be done with capability CAP_SYS_ADMIN (root user))

3.1 Why a Kernel-Patch?

The vanilla 2.6 Linux kernel does not provide a mechanism to trace syscalls in the desired way, nor does it contain the capability to track processes and generate an audit trail. Due to this lack of functionality the Linux kernel needs to be patched. The patch enhances internal kernel structures to keep track of the process and provides an interface to the user space by defining I/O control commands and a device file.

Beside file–system DAC controls of the audit device file the kernel patch restricts access by verifying if the caller of an I/O control command has the capability CAP_SYS_ADMIN.

3.2 How can a Process be attached/detached to/from LAuS?

A process can only attach itself to the audit subsystem and only if it has root (CAP_SYS_ADMIN) privileges. Attaching is done by either directly using I/O control commands or by using LAuS library functions. Several attributes, such as the "Login ID" and the "Audit ID" are bound to the attached process.

Whenever an audited process forks a child process, the child process inherits some attributes of the parent process to make the audit trail continuous.

Likewise, the only instance that can detach a process is the process itself, and only if it has root privileges (CAP_SYS_ADMIN). When detaching, all session information (such as the the Login ID and Audit ID) is lost.

Another way of detaching is to exit. Whenever a process terminates/aborts it will be detached from the audit subsystem, too.

In addition, a process is permitted to suspend and resume auditing. Again, this is achieved through I/O control commands to the audit subsystem, and requires administrative privilege (CAP_SYS_ADMIN). This functionality is for the benefit of trusted applications that wish to generate a single high-level audit event describing their actions, instead of several system call events.

The major difference between suspending and detaching is that the former retains all session information, including the "Login ID" and "Audit ID". The suspend flag is not inherited to child processes, that is, if a process suspends auditing and forks a new child process, that child will be subject to auditing as usual.

A trusted application such as the `passwd` utility, for instance, suspends auditing before updating the password database, and generates a single record indicating the (attempted) password change afterwards.

3.3 How will Events be generated?

There are two kinds of sources for an audit event, the kernel and user applications. The main source is the kernel space. System calls, file access and network layer actions are handled by the kernel. Netlink operations are logged on completion. System call arguments are normally logged inside the call at the beginning of the function, return codes are logged at the end of the system call.

- *adjtimex(2)* arguments will differ at beginning and end of the call.
- *chroot(2)* needs to process the filename in the context of the original file–system root, not the changed one after the system call.
- *execve(2)* never returns, and the process image will be gone as soon as the syscall is finished.
- *ioctl(2)* arguments will differ at beginning and end of the call.
- *rename(2)* needs to save the first argument before the call, it will be invalid after completion of the syscall.
- *unlink(2)* needs to process the filename argument before the file is deleted.

Access to file–system objects is logged indirectly through hooks in the Virtual File System (VFS).

User applications have the option to generate their own, more abstract, audit records, as a replacement or enhancement for the low-level syscall-based records. For example, the `passwd(1)` utility should write a single notification of the password change instead of several low-level operations on the password files. To do this, trusted programs can suspend syscall auditing and send user messages to the kernel using the audit device. The kernel will add its headers and attributes, then pass the message on to the audit daemon through the device file.

Every event generated by the kernel contains information on the process on behalf of which the kernel generates the event, including the current UID, GID, the "Login ID" and "Audit ID", etc. This fixed portion is followed by a variable data portion, depending on the message type.

Event messages are placed into a queue, from where they can be retrieved by the audit daemon through the `read` system call, one record at a time. If the length of the queue exceeds a certain prespecified limit (adjustable via the `sysctl` kernel interface), any processes trying to generate new events will be blocked until there is space in the queue again. The default maximum size of the queue is 1024 entries with 8 KB per entry.

3.3.1 Kernel Source

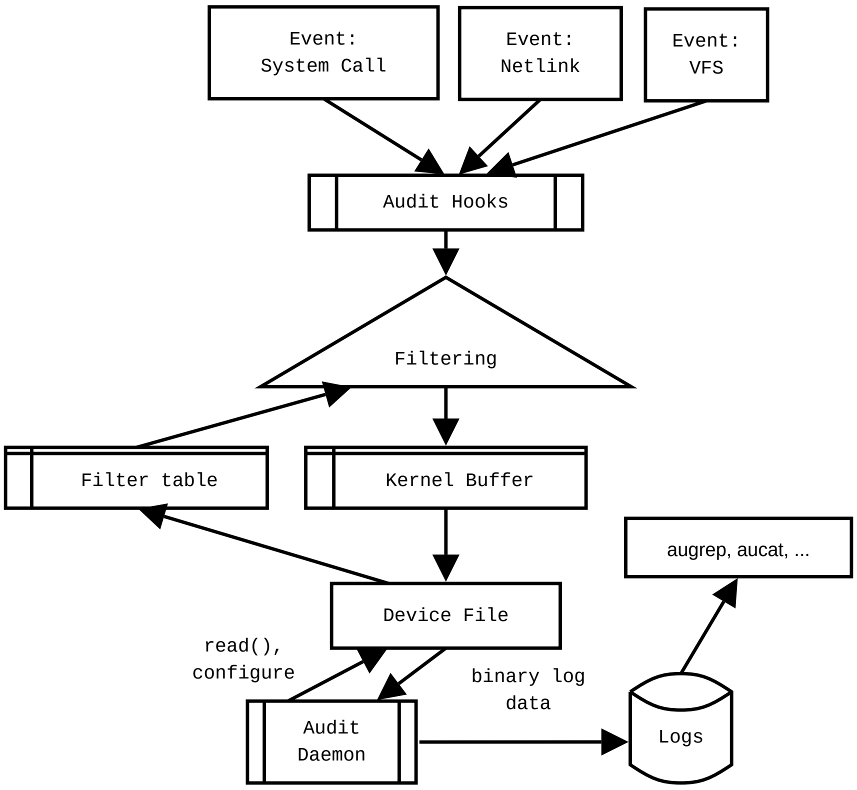

Figure 3.1: Data Flow: Kernel Sources

The kernel patch creates several hooks for monitoring process creation/termination, VFS usage, and system calls entry/return, as well as a hook to track modifications of the system's network configuration.

System Calls

As stated before a sysem call will be intercepted at the beginning of the code and at the various return points (different errors). System call events will be generated for every traced process as long as the filter policy does not discard it. The filter policy can be a simple yes/no statement, but complex Boolean expressions involving properties of the process, as well as the system call arguments, are possible, too.

If the system call passes the filter rules, an audit event will be generated. This event data includes information about the process, system call number, the return value (outcome), and a TLV (tag/length/value) encoded representation of the system call arguments, where applicable. (For instance, the argument data to a number of ioctl calls are included, but data passed to the `write` system call is generally not included).

Filesystem Hooks

To log the usage of filesystem objects the kernel patch relies on a prerequisite patch to the VFS code modifying functions like `open`, `truncate`, `chdir` and so on. The modifications of these functions are similiar to those of the system calls; intercept functions are inserted at the beginning and end of a monitored VFS function.

Netlink Sockets

The Linux kernel network code can be controlled either by using the *ioctl(2)* system call of by using a netlink socket. The first case is handled as described above in sub-section “System Calls“. The latter case needs special handling. To become aware of netlink messages the kernel patch needs to apply another hook in the kernel. LAuS only observes netlink routing messages because these are the ones we are interested in. To get the result of the message processing the audit hook is triggered right after the message had been processed. The message data, message length and the outcome will be logged.

Process Creation and Termination

The audit subsystem can generate audit events for process creation (including processes generated by `fork` and `clone`, but also for kernel threads), and process termination. For both events, filter policies can be configured to select just specific events (such as processes exiting due to a signal).

3.3.2 User Source

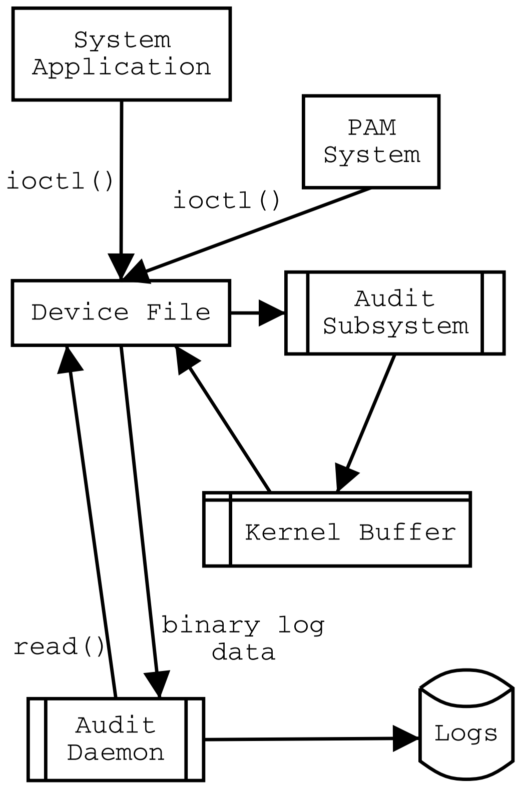

Figure 3.2: Data Flow: User Sources

In addition to the kernel, user space applications should be able to generate their own, more descriptive, audit records. This type of records is called "Audit User Messages". Two types of user applications need this special feature:

a. applications that authenticate users and/or change privileges

b. applications that change the configuration of the system

The first group of applications can be served by a special PAM library and a PAM module. The PAM library and the module attach the current process and set various attributes like the "Login ID", the terminal name, hostname, IP address and alike through a special „Audit Login Message„. The PAM module can serve as an authentication, account or session module. It is used as workaround for applications that handle authentications apart from PAM but use the PAM framework for other tasks.

The latter group of applications needs to be modified manually to handle the LAuS interface to the kernel and to send the "Audit User Messages".

The PAM Framework

The PAM module is used together with the modified PAM library patch to activate the audit subsystem for the current application. The module is responsible for the following tasks:

- open the audit device file
- if configured to do so, detach the current audit data
- attach the current process to the audit subsystem
- close the audit device file

The PAM Library is patched to write audit logs for success and failure returned by the PAM module stacks called on behalf of applications. The library framework is responsible for the following tasks:

- open the audit device file
- emit an “Audit User Message“ indicating success or failure
- on successful authentication, set the login UID for the process and emit an “Audit Login Message“
- close the audit device file

The kernel does not care about the format of the “Audit User Messages“, it just adds the attributes and header to it and puts it in the audit record queue.

All system applications that handle authentication for changing user privileges are linked against the PAM library. Therefore the PAM library provides a central point for handling LAuS operations.

Enhanced System-Applications

All system applications that change the system configuration need to be modified to notify the SO about the changes they made. This does not need system call auditing, so the trusted application can suspend auditing and perform their own logging. To accomplish this task just a few lines of code need to be added:

1. open LAuS interface
2. suspend auditing
3. format user message and send it to the kernel
4. close LAuS interface

3.4 What Information will be kept per Event?

Additional information is generated and stored with each event. The following list gives an overview (please note: some informations are accessed indirectly by referencing the “Audit ID“):

- Timestamp: Every audit record is timestamped
- Login ID: User ID of the user authenticated by the system
- Audit ID: unique 32 bit identifier
- Login Message:
  - Hostname: Remote host name in case of remote login
  - IP Address: IP address of remote host in case of remote login
  - Service: Name of service that authenticates the user
- Text Message:
  - arbitrary User-Text
- System Call:
  - System call name
  - Arguments
  - Result/Outcome

3.5 How will a unbroken Audit Trail be guaranteed?

To guarantee a continuous audit trail, three mechanism will be used:

- Putting audited processes to sleep when the audit record buffer is full or something is wrong with the log file.
- pre-allocated bin files
- or alternatively: monitoring disk-space while in stream- or file- mode and notify the SO if threshold is reached.

3.6 How does the Audit Record reach the User-Space?

First the audit daemon has to register itself to LAuS to receive all audit records. The audit records themselves are written to an internal queue and can be read, one at a time, from there by invoking the `read` system call on the audit device file. The audit daemon is the only process that is able to read these records. Every record read will be deleted from the queue to free memory for new ones.

3.7 How will the Audit Record be written?

After the audit daemon reads an audit record from the device file it will add another header containing just a timestamp. The payload data will not be processed in any way. Therefore the audit log just contains the time and the binary data that was directly read from the kernel.

3.8 What about post-processing the Audit Record?

Tools like `aucat` use various library functions to parse the binary audit log and output it in a human readable form. These library calls can be used by every application that wants to postprocess the log files.

3.9 Who can configure what in which way?

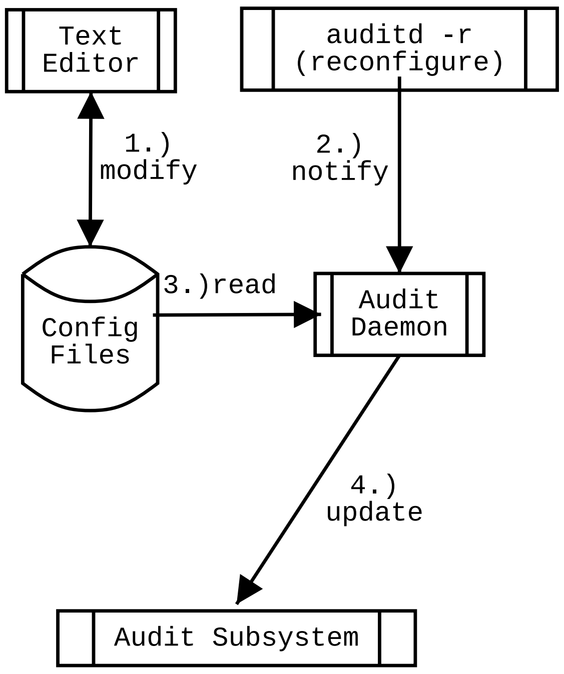

Figure 3.3: Data Flow: Configuration

By using the DAC controls of the file–system only the users (typically root) with capability CAP_DAC_OVERRIDE or CAP_DAC_READ_SEARCH are allowed to access and modify the configuration file of LAuS. The only component of LAuS that uses configuration files is the audit daemon. The audit daemon needs a main configuration file for defining thresholds and corresponding actions etc, and two files for defining filter rules and filter object sets.

These configuration files need to be modified directly by using a text editor and can be made effective by running `auditd -r`. This command sends a reload message (HUP signal) to the running auditd process, which will re-read the configuration and reload it into the kernel. DAC controls ensure that only the root user is able to modify these files and use `auditd -r`, additionally the audit subsystem only accepts messages generated by user root.

3.10 How is the configuration transferred to the Kernel?

The audit daemon reads the configuration files, parses them and sends the filter rules to the kernel by using a special I/O control command. The filter rules are part of the kernel now and can only be modified or cleared by a user with sufficient administrative privilege (CAP_SYS_ADMIN).

Chapter 4

Low Level Design

4.1 LAuS Components

The core component of LAuS are two kernel patches to enable file–system hooks and system call logging (partially based on the former), filtering, checking network traffic and keeping track of user activities. In addition, it contains an audit daemon to handle kernel messages, several command line tools, LAuS API libraries, a modified PAM subsystem, a PAM module, and modified system applications. The following diagram is an overview of the LAuS components:

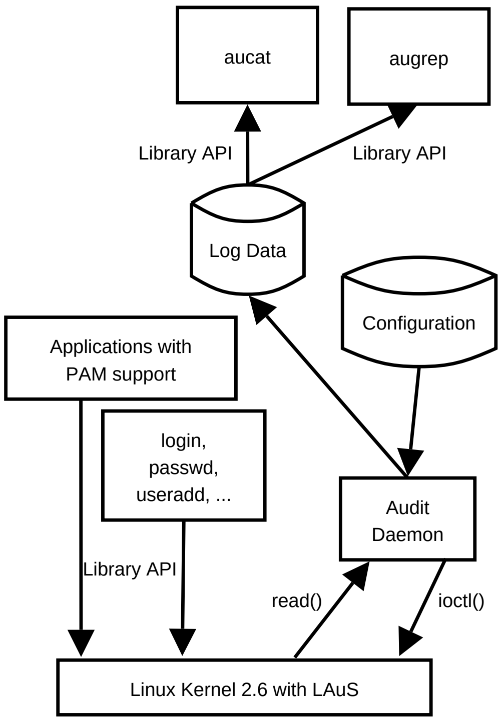

Figure 4.1: LAuS Overview

4.1.1 Kernel Patch

The native Linux kernel does not contain any mechanism to monitor system calls and to keep track of user activities. Therefore the Linux kernel has to be enhanced to provide the SO with an audit trail.

The kernel patch modifies the process task structure for storing additional information/attributes, adds intercept functions and an additional flag to the ptrace framework, provides an interface to the user space, and applies filter policies. All these tasks will be described in the following subsections.

Login ID

In order to fulfill the CAPP requirements, the kernel must be modified to track the "Login ID" for each process. The "Login ID" is part of the `Audit Login Message` that is sent to the kernel and includes information like hostname, IP address, terminal name, and the name of the executable. The "Login ID" is stored in the structure `aud_process` and should not be confused with the "Audit ID". The "Login ID" is the numerical Unix UID of the user logged in, and the "Audit ID" is a unique session identifier. Therefore, there can be multiple sessions with the same "Login ID" when a user has logged in several times simultaneously, but each will have a different "Audit ID".

Audit ID

In addition to the "Login ID", an "Audit ID" is stored in the structure `aud_process` to identify the trail of a process tree. The "Audit ID" is unique for each session and will be assigned when the session initiator (i.e. `login` or `sshd`) attaches to the audit subsystem. If the process spawns a child process, this ID gets inherited, and therefore stays the same for all processes launched as part of this session.

Task Structure

The process task structure as defined in `linux-2.6.4/include/linux/sched.h` is enhanced by a void pointer.

```
#if defined(CONFIG_AUDIT) || defined(CONFIG_AUDIT_MODULE)
        void *audit;
#endif /* CONFIG_AUDIT */
```

This void pointer is used by the audit device driver to point to audit related data. The audit driver manages the following data for every audited process:

```
struct aud_process {
        struct list_head        list;
        uid_t                   login_id;
        unsigned int            audit_id;
        /* Auditing suspended? */
        unsigned char           suspended;
};
```

If an audited process forks, the child process will receive a fresh aud_process structure, and the audit_uid and audit_id fields will be copied from the parent process. The suspended field is initialized to zero, to ensure that new processes launched from a trusted program start with auditing active by default even if the parent has suspended it.

Audited System Calls

LAuS catches only a subset of syscalls provided by the Linux kernel but all syscalls needed for CAPP/EAL4.

Before we began with a list of traced system calls we should look at an example of a patched settimeofday system call.

```
--- /usr/src/linux-2.6.4-29/kernel/time.c
+++ 2.6.4-29-LAuS/kernel/time.c
@@ -28,6 +28,7 @@
 #include <linux/timex.h>
 #include <linux/errno.h>
 #include <linux/smp_lock.h>
+#include <linux/audit.h>
 #include <asm/uaccess.h>

 /*
@@ -74,13 +75,14 @@
        struct timespec tv;

        if (!capable(CAP_SYS_TIME))
-               return -EPERM;
+               return audit_intercept(AUDIT_settimeofday, NULL, NULL),
                                       audit_result(-EPERM);
        if (get_user(tv.tv_sec, tptr))
                return -EFAULT;

        tv.tv_nsec = 0;
+       audit_intercept(AUDIT_settimeofday, &tv, NULL);
        do_settimeofday(&tv);
-       return 0;
+       return audit_result(0);
 }
```

```
#endif
```

Syscall Name	**analyzed?**	**needed?**
access	yes	yes
adjtimex	yes	yes
brk	yes	yes
capset	yes	yes
chdir	yes	yes
chmod	yes	yes
chown	yes	yes
chown32	yes	yes
clone	yes	yes
create	yes	yes
create_module	yes	yes
delete_module	yes	yes
execve	yes	yes
fchmod	yes	yes
fchown	yes	yes
fchown32	yes	yes
fork	yes	yes
fremovexattr	yes	yes
fsetxattr	yes	yes
init_module	yes	yes
ioctl	yes	yes
ioperm	yes	yes
iopl	yes	yes
ipc (msgctl, msgget, semctl, semget, shmat, shmctl, shmget)	yes	yes
lchown	yes	yes
lchown32	yes	yes
link	yes	yes
lremovexattr	yes	yes
lsetxattr	yes	yes
mkdir	yes	yes
mknod	yes	yes
mount	yes	yes
open	yes	yes
ptrace	yes	yes
removexattr	yes	yes

rename	yes	yes
rmdir	yes	yes
semtimedop	yes	yes
setfsgid	yes	yes
setfsgid32	yes	yes
setfsuid	yes	yes
setfsuid32	yes	yes
setgid	yes	yes
setgid32	yes	yes
setgroups	yes	yes
setgroups32	yes	yes
setregid	yes	yes
setregrid32	yes	yes
setresgid	yes	yes
setresgid32	yes	yes
setresuid	yes	yes
setresuid32	yes	yes
setreuid	yes	yes
setreuid32	yes	yes
settimeofday	yes	yes
setuid	yes	yes
setuid32	yes	yes
setxattr	yes	yes
socketcall (bind)	yes	yes
swapon	yes	yes
symlink	yes	yes
truncate	yes	yes
truncate64	yes	yes
umask	yes	yes
unlink	yes	yes
utime/utimes	yes	yes
vfork	yes	yes

Filesystem Hooks

To trace the access to file–system objects several functions in the VFS code were intercepted similar to the system call hooks. The intercept functions are wrapper inside C macros like `FSHOOK_BEGIN` and `FSHOOK_END`. The description of `FSHOOK_BEGIN` will serve as an example here.

1. copy arguments

2. check who is registered for this call

2. pass information about call to registered instance

Handling I/O Control Messages

For specific I/O control messages, the audit module will intercept the data passed by the caller and include it in the audit event. For all other I/O control messages, data is not included in the audit event.

The following network-related I/O control messages have data included in the audit event:

SIOCADDRT:	add routing table entry
SIOCDELRT:	delete routing table entry
SIOCSIFLINK:	set iface channel
SIOCSIFFLAGS:	set flags
SIOCSIFADDR:	set PA address
SIOCSIFDSTADDR:	set remote PA address
SIOCSIFBRDADDR:	set broadcast PA address
SIOCSIFNETMASK:	set network PA mask
SIOCSIFMETRIC:	set metric
SIOCSIFMEM:	set memory address (BSD)
SIOCSIFMTU.	set MTU size
SIOCSIFNAME.	set interface name
SIOCADDMULTI.	Multicast address lists
SIOCDELMULTI.	
SIOCSIFHWADDR.	set hardware address
SIOCSIFENCAP:	
SIOCSIFSLAVE:	
SIOCSIFPFLAGS.	set/get extended flags set
SIOCDIFADDR:	delete PA address
SIOCSIFHWBROADCAST:	set hardware broadcast addr
SIOCSIFBR:	Set bridging options
SIOCSIFTXQLEN:	Set the tx queue length
SIOCDARP:	delete ARP table entry
SIOCSARP:	set ARP table entry
SIOCSIFMAP:	Set device parameters
SIOCADDDLCI:	Create new DLCI device
SIOCDELDLCI:	Delete DLCI device

Handling IP Device and Routing Changes

The Linux kernel supports two mechanisms for configuring IP network devices, and IP routing:

- through *ioctl(2)*
- through `AF_NETLINK` sockets

I/O control messages are handled by identifying the messages we're interested in, and copying the data that comes with them. Netlink messages are the more advanced mechanism of network configuration, and is used by utilities such as `ip(8)`. Netlink messages are sent through sockets of type `AF_NETLINK`, where the destination is identified by numeric IDs such as `NETLINK_ROUTE`. Alternatively, netlink messages can be delivered to specific processes.

The only recipient ID relevant to our TOE is `NETLINK_ROUTE`. Delivery to specific processes is not relevant to auditing network configuration. `CAP_NET_ADMIN` privilege is required to create a netlink socket capable of receiving/sending `NETLINK_ROUTE` messages. A netlink message consists of one or more parts, each comprising a header of type `struct nlmsghdr`, followed by data specific to the recipient ID. The common data part of all `NETLINK_ROUTE` messages consists of a `struct rtgenmsg` containing the address family.

The IPv4 routing code receives these messages by registering a handler for `PF_INET` with the rtnetlink component. Similarly, the IPv6 code registers a handler for `PF_INET6`.

The audit code taps into the rtnetlink code, specifically into `rtnetlink_rcv_skb` which takes care of delivering `NETLINK_ROUTE` messages through these handlers. The function delivers each portion of the message individually, and sends the outcome of the code back to the calling sockets. The call hooks to the audit module are invoked after the netlink message has been processed, passing the message itself, the message length and the outcome for inspection by the audit module.

If the audit module decides to generate an audit event for the netlink message, the event generated includes the contents of the message and the outcome.

Device File

To enable bidirectional communication between user space and kernel space LAuS provides a character device file. Communication happens via *ioctl(2)* calls and by using *read(2)*. The latter function call is used to read audit records from kernel buffers and i.e. write them to disk.

The format of the audit record will be explained in detail in section "Contents of Audit Record", the *ioctl(2)* commands are explained in the next subsection.

The LAuS device file is a character device named `/dev/audit` and has the major number 10 (misc devices) and minor number 224.

LAuS I/O Messages

The following table shows the *ioctl(2)* commands, their arguments, and their description.

Command	**Argument**	**Description**
AUIOCATTACH	none	Attach current process to audit subsystem
AUIOCDETACH	none	detach current process from audit subsystem
AUIOCSUSPEND	none	Suspend auditing for current process
AUIOCRESUME	none	Resume auditing for current process
AUIOCCLRPOLICY	none	Clear policy
AUIOCSETPOLICY	struct audit_policy	Add policy
AUIOCCLRFILTER	none	Clear filter
AUIOCSETFILTER	struct audit_filter	Add filter
AUIOCIAMAUDITD	none	Register current process as audit daemon
AUIOCSETAUDITID	none	Set Audit-ID
AUIOCLOGIN	struct audit_login	Generate login message
AUIOCUSERMESSAGE	struct audit_message	Generate user-specified message

Filter

To reduce the I/O load and to reduce the amount of logging data the kernel is able to perform filtering by using predicates and logical operations. Basic predicates can be combined to user defined and more complex predicates like the following example illustrates:

```
predicate is-one-or-two = eq(1) || eq(2);
```

The predicates can be used by defining a filter or by attaching the predicate to a syscall.

```
filter uid-is-one-or-two = is-one-or-two(uid);
...
syscall sleep = is-one-or-two(arg0);
```

The filter is used to bind the predicate to a so called target (syscall argument, process property, syscall result, etc.)

To handle a class of objects more easily the audit filter allows to specify a so called ‘set‘.

```
set sensitive = { /etc, /root, /usr }
...
predicate is-sensitive = prefix(@sensitive);
```

The example above illustrates the use of sets. A set can be referenced by a leading ‘@‘ sign. The man page `audit-filter.conf(5)` gives a more detailed description the filtering scheme.

4.1.2 Audit Daemon

The audit daemon performs the following functions

- announce himself to the audit-subsystem
- turns kernel auditing on and off
- sends the audit filter policy to the kernel audit-subsystem
- reads the audit records from the device file
- writes the audit records to the disk (file-, stream-, bin-mode)
- monitors the current state of the system for potential audit record loss
- notifies the system administrator via syslog in case of impending audit data loss

The audit daemon provides three ways of writing audit records to disk. The choice of which method to use is configurable by the administrator. The choices are ‘file mode‘, ‘bin mode‘ and ‘stream mode‘. In file mode, data is written pretty much the same way as syslogd(8) does, i.e. records are appended to a file that is allowed to grow arbitrarily until culled by the administrator. Culling may happen either by truncating the file, or by moving it aside and sending the daemon the hangup signal (SIGHUP).

If any error occurs when writing to the file, auditd will go into error mode. Depending on the configuration, auditd can perform different actions in response to an error, ranging from totally ignoring it to halting the system.

If no log destinations are specified in audit.conf, file mode will be used to write the audit trail to `/var/log/audit`.

Streaming mode is pretty much like file mode, except that data is sent to an external command on standard input. This allows forwarding audit data to other hosts via arbitrary mechanisms (including stunnel, ssh, etc).

In stream mode, an audit record stream is piped to an user defined program for post-processing.

In bin mode, multiple fixed length files are maintained with a pointer to the current location. The audit records are written until the current file has reached it maximum capacity, then the next file is used until it reaches its maximum capacity, cycling round-robin through the available files, finally reusing the first file. This allows the administrator to specify the maximum disk space that audit records will ever take. A notification program can be used to save each full bin file for long-term storage.

4.1.3 Audit Tools

The user space tools consist of `aucat`, `augrep`, and `aurun`. `aucat` reads the audit log files and outputs the records in human readable format. `augrep` performs a similar function but it allows the administrator to optionally filter the records based on user, audit id, outcome, system call, or file. `aurun` can be used as a wrapper to start applications, like `Apache`, and attach them to the audit subsystem without modifying the application's source code.

The usage of the tools is described in the corresponding online man pages distributed with the LAuS library: `aurun(8)`, `aucat(1)`, and `augrep(1)`.

4.1.4 Enhanced PAM Library and the PAM Module

The modified PAM library and the PAM LAuS module work together to set up the auditing environment.

A complication here is that not all applications use the PAM framework in exactly the same way, for example `sshd` bypasses PAM authentication when the user authenticates using a private key instead of a password.

Also, there are two conflicting requirements concerning the attached audit information. On the one hand, actions done by an administrator must be audited with the admin's original non-root login UID, including for processes started using `su`. On the other hand, if the administrator restarts a system daemon such as `sshd`, users who log in using that restarted daemon must receive a fresh login record, and not have their actions audited with the data of the administrator who restarted the service.

Therefore, some flexibility in configuring the PAM system is required.

The module `pam_laus.so` is responsible for activating auditing for the current process. It calls `laus_init()` and `laus_open()` to open the audit device file, then `laus_attach()` to attach the current process to the audit subsystem and `laus_setauditid()` to assign a fresh audit session ID.

As a special case, if the module flag `detach` is set, a call to `laus_detach()` is done before the call to `laus_attach()` to disassociate any previously attached audit data from the process. This flag MUST be used in the PAM configuration file of services such as `sshd` or `ftpd` that require a clean environment for newly logged-in users. It MUST NOT be used for reauthenticating services such as `su` or screen savers, where the currently attached audit data remains valid for the new process.

The PAM library implements a central intercept hook `_pam_auditlog()` that is called at the end of each stack of `auth`, `account`, `session` and `password` modules. An Audit User Message is written to the audit log indicating success or failure as determined by the module stack's returned value.

The PAM configuration for each service MUST ensure that the `pam_laus.so` module is run in every case before control is given to the user. This can be done in any one of the `auth`, `account` or `session` stacks, but the application code MUST be verified to ensure that this stack is used in every case. For example, `sshd` always runs the `account` stack, but bypasses the `auth` stack in the case of public key authentication.

Note that the audit functions require CAP_SYS_ADMIN capabilities (usually equivalent to root rights), so if a stack is not run as root, they will fail. For example, `sshd` runs the `session` stack with the logged-in user's rights, so putting the `pam_laus.so` module in that path will not work.

4.1.5 Enhanced System Applications

Applications like login or passwd can write arbitrary text messages to the audit daemon through the kernel by using the ioctl command AUIOCUSERMESSAGE. This enables security relevant system applications to write short and descriptive messages into the audit logs without using syscall logging.

The following list shows all instrumented applications:

- crond
- crontab
- atd
- at
- useradd
- userdel
- usermod
- groupadd
- groupdel
- groupmod

- gpasswd
- passwd
- rpasswdd
- chage

4.2 LAuS Configuration

Currently just the audit daemon has configuration files. All other components are simple enough to configure via command line arguments.

4.2.1 Audit Daemon

The audit daemon needs three configuration files. The main config file (`audit.conf`) is used to set the path to the filter rules, to define disk space thresholds and alike. The files `filter.conf` and `filesets.conf` (not mandatory, just used to ease configuration) are used for filtering.

Please refer to the online man pages *audit.conf(5)*, *audit-filter.conf(5)*, and *audit-filesets.conf(5)* for more details.

4.3 LAuS Log Files

In the default configuration the audit daemon writes its log data to `/var/log/audit`. The log data can be read with the command `aucat(1)`.

4.3.1 Contents of Audit Record

The audit record written to the device file depends on the type of message (enter syscall, leave syscall). The audit record will include the major and minor version number of LAuS and a flag for specifying the byte order.

Please refer to the *laus-record(7)* man page for a description of the data structures used.

4.3.2 Raw Log Format

The raw log format just contains the binary data from the kernel and a header to add the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

4.3.3 Cooked Log Format

By using the function `laussrv_process_log()` of the server API library it is possible to obtain the timestamp and raw kernel data via a callback function. The callback function can use the various print functions of the server API library to output the data in human readable informations. Example:

```
25 Jun 03 16:56:50     root     8 LOGIN: uid=0, terminal=/dev/pts/2,
                                  executable=/usr/bin/aurun
25 Jun 03 16:56:50     root     8 open("/etc/shadow", 32768, 0) = 3
25 Jun 03 19:04:12     okir    11 LOGIN: uid=100, terminal=/dev/tty1,
                                  executable=/bin/login
25 Jun 03 19:04:12     okir    11 login: Authentication succeed for
                                  User 'okir' (100)
25 Jun 03 19:04:32     okir    11 open("/etc/shadow", 32768, 0) =
                                  Permission denied (error 13)
```

This trail shows a process started via `aurun`, which opened the shadow file for reading, and a user logging via `/bin/login`, and trying to open the shadow file as well.

Appendix A

Abbreviations

BSI	Bundesamt fuer Sicherheit in der Informationstechnik
BSM	Basic Security Module
CAPP	Controlled Access Protection Profile
CC	Common Criteria
CERT	Computer Emergency Response Team
DAC	Discretionary Access Control
DoS	Denial–of–Service
EAL	Evaluation Assurance Level
FIFO	First In, First Out; Named Pipe; local Interprocess Communication
GNU	GNU's Not Unix!, Projekt of the *Free Software Foundation*
GUI	Graphical User Interface
IDMEF	Intrusion Detection Message Exchange Format
IDS	Instrusion Detection System
IP	Internet Protocol, s. RFC–791 [3]
LAuS	Linux Audit-Subsystem
LKM	Loadable Kernel Modul
PAM	Pluggable Authentication Module
SO	Security Officer

SQL	Structured Query Language
SSL	Secure Socket Layer, Encryption on presentationlayer
Syslog	native Unix Logging System
TCP	Transmission Control Protocol, s. RFC–793 [3]
UDP	User Datagram Protocol, s. RFC–768 [3]
UML	Unified Modeling Language
VFS	Virtual Filesystem, abstrace layer of filesystem
XML	Extensible Markup Language

Appendix B

List of Figures

Appendix C

Bibliography

[1] D. Curry, H. Debar, Intrusion Detection Message Exchange Format — Data Model and Extensible Markup Language (XML) Document Type Definition, IDWG, February 2002

[2] LibIDMEF, `http://www.silicondefense.com/idwg/libidmef/index.htm`

[3] RFC Datenbank, `http://www.rfc-editor.org/`

[4] GNU Free Documentation Licence, `http://www.gnu.org/copyleft/fdl.html`

[5] CAPP Version 1d, `http://www.radium.ncsc.mil/tpep/library/protection_profiles/CAPP-`