Linux Audit-Subsystem Design Documentation

Version 1.2 RC6

IBM/SuSE Confidential until LAuS Release

Changelog

Version	Date	Authors	Reviewer	Changes, Problems, Notes
0.1	2003−05−17	Emily		− Based on Janak Desai's design template and Thomas Biege's design abstract and API specification
0.2	2003−05−19	Emily, Dan		− Added a section listing the system calls that are being audited and another section for valid ioctls(). Updated picture and some info with new material from today's phone call.
0.3	2003−05−22	Thomas		− try to merge 0.1 with 0.2 − try to answer some questions − reflect current design
0.4	2003−05−26	Thomas		− added description of user space tools − added new log format
0.5	2003−06−03	Thomas		− added auditd config file − explain which tool uses which API call − describe audit record
0.6	2003−06−04	Thomas		− add section to describe the point all syscalls need to go through − add filter config file example
0.7	2003−06−13	Thomas		− filled table for CAPP requirements − completed audited syscalls
1.0	2003−06−25	Thomas		− migrated to LaTeX − rewrote everything − changed API description − completed Low Level Design − added High Level Design
1.1	2003−06−27	Thomas		− applied corrections made by Olaf − added more information to answer afx' questions − added 3 pictures to illustrate data-flow − completed bib. − updates ToDo
1.2 rc2	2003−08−01	Thomas		− try to answer open questions − changes in respect to new PAM modifications − describe audit tools − mention action "shutdown" as panic option to execute when we run out of disk space − describe new kernel hook design − struct aud_message changes due to timestamp creation in kernel space − added Klaus' PAM description − added AUTH_failure − removed AUDIT_shutdown − changed origin of AUDIT_start, AUDIT_stop
1.2 rc3	2003−08−06	Thomas		− removed confidential flag − completed command line tool description
1.2 rc4	2003−08−12	Thomas		− removed tag: FILE_chpriv, FILE_fchpriv, FILE_facl, FILE_fchmod, FILE_fchown

				– added tag: FILE_loginid – expanded tag: FILE_truncate, FILE_owner, FILE_mode
1.2 rc5	2003−08−18	Thomas		– added new filter.conf file with audit-tags and device major/minor filter – added warning for overwriting /var/log/audit – glibc API vs. kernel API – added warning because of overwriting /var/log/audit – added description for audbin
1.2 rc6	2003−08−27	Thomas		– added note about name collision due to using /dev/audit – updated lists of ioctl(2) commands
1.2 rc6	2003−09−01	Thomas		– updated filter.conf and filesets.conf – updated lists of syscalls
1.2 rc6	2003−09−17	Thomas		– updated list of audit tags

Copyright Notes

Abstract

This paper describes the development of the Linux Audit-Subsystem (LAuS), its components, its configuration and its CAPP compliance. LAuS was developed by SuSE Inc. to make Linux more secure and to attain the EAL3 certificate.

Contents

Chapter 1

Introduction

The purpose of this document is to describe the development of the Linux Audit-Subsystem. LAuS is one important part for the Common Criteria evaluation for SuSE Enterprise Server 8. The desired evaluation level is EAL3.

Additionally this document serves as a communication platform for the development teams of IBM and SuSE to describe low and high level design, clarify design decisions and answer open questions.

Chapter 2

CAPP Requirements

While developing and designing LAuS the CAPP version 1d of the Information Systems Security Organization [4] has to be kept in mind, otherwise the development of LAuS may become useless.

2.1 Audit Data Generation FAU_GEN.1

Section	**Component**	**Event**	**How does LAuS meets this?**
5.1.1	FAU_GEN.1	Start-up and shutdown of the audit functions	Events: AUDIT_start, AUDIT_stop
5.1.2	FAU_GEN.2	None	
5.1.3	FAU_SAR.1	Reading of information from the audit records.	Events: FILE_read, FILE_open
5.1.4	FAU_SAR.2	Unsuccessful attempts to read information from the audit record	Like FAU_SAR.1 (FILE_read, FILE_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: FILE_open, FILE_write, AUDCONF_alter, AUDCONF_reload
5.1.7	FAU_STG.2	None	
5.1.8	FAU_STG.3	Actions taken due to exceeding of threshold.	Event: AUDIT_disklow
5.1.9	FAU_STR.4	Actions taken due to the audit storage failure	Event: AUDIT_diskfail
5.2.1	FDP_ACC.1	None	

5.2.2	FDP_ACF.1	All requests to perform an operation on an object covered by SFP.	Events: FILE_mode, FILE_owner, FILE_link, FILE_mknod, FILE_open, FILE_create, FILE_rename, FILE_truncate, FILE_unlink, FS_rmdir, FS_mount, FS_umount, MSG_owner, MSG_mode, MSG_delete, MSG_create, SEM_owner, SEM_create, SEM_delete, SEM_mode, SHM_create, SHM_delete, SHM_owner, SHM_mode
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_pwchange, AUTH_success, AUTH_failure
5.3.3	FIA_UAU.1	All use of the authentication mechanism.	Events: AUTH_pwchange, AUTH_success, AUTH_failure
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_pwchange, AUTH_success, AUTH_failure
5.3.6	FIA_USB.1	Success and failure of binding user security attributes to a subject (e.g. success and failure to create a subject).	Events: PROC_execute, PROC_realuid, PROC_auditid, PROC_loginid, PROC_setuserids, PROC_realgid, PROC_setgroups

5.4.1	FMT_MSA.1	All modifications of the values of security attributes.	Events: PROC_execute, PROC_realuid, PROC_auditid, PROC_loginid, PROC_setuserids, PROC_realgid, PROC_setgroups, PROC_privilege
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.	Events generated: FILE_open, FILE_write
5.4.3	FMT_MTD.1	All modifications to the values of TSF data.	Events: FILE_open, FILE_write, AUDCONF_alter, AUDCONF_reload
5.4.4	FMT_MTD.1	All modifications to the values of TSF data.	Events: FILE_open, FILE_write, FILE_read, AUDCONF_alter, AUDCONF_reload
5.4.9	FMT_SMR.1	Modifications to the group of users that are part of a role.	Event: PRIV_userchange
5.4.9	FMT_SMR.1	Every use of the rights of a role (Additional/Detailed)	Syscall: setuid(2) (???)
5.5.1	FPT_AMT.1	Execution of the test of the underlying machine and the result of the test.	Audit events for the abstract machine testing tool. This might be also handled by a log file from the diagnostics program. Events: ADMIN_amtu
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: SYS_timechange

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. In 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 will provide a user space tool, `aucat`, that will translate 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 will be protected by DAC controls so that only an authorized administrator will be able to read the logs. The audit tools will also be protected by DAC controls so that only authorized administrators can invoke the tools.

2.5 Selectable Audit Review FAU_SAR.3

LAuS will provide a user space tool, `augrep`, that will allow the administrator to filter the audit records to only display requested events. The administrator will be 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 will provide the administrator the ability to select the events to audit. This will be done by the administrator editing the filter configuration file of the audit daemon and then using the `aucfg` tool 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 will prevent 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 disc 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.

2.9 Prevention of Audit Data Loss FAU_STG.4

To avoid the loss of data two so called "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 second bin file is used for storing audit records. If the command fails (i.e. exits with a non-zero exit status), the SO will be notified via 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.

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.

2.13 Mapping Events

Event	**Syscall/Function/Program**
AUDIT_start	audit module
AUDIT_stop	audit module
AUDIT_disklow	audit daemon
AUDIT_diskfail	audit daemon
AUDCONF_alter	audit daemon
AUDCONF_reload	audit daemon
AUTH_pwchange	passwd
AUTH_success	su, login, sshd, ftpd, PAM, ...
AUTH_failure	su, login, sshd, ftpd, PAM, ...
FILE_mode	chmod(2), fchmod(2)
FILE_owner	chown(2), lchmod(2), chown32(2), lchmod32(2), fchown(2)
FILE_link	link(2), symlink(2)
FILE_mknod	mknod(2)
FILE_open	open(2)
FILE_create	create(2), open(2)
FILE_rename	rename(2)
FILE_truncate	truncate(2), truncate64(2), ftruncate(2), ftruncate64(2)
FILE_unlink	unlink(2)
FS_rmdir	rmdir(2)
FS_mount	mount(2)
FS_umount	umount(2), umount2(2)
MSG_owner	ipc(2)
MSG_mode	ipc(2)
MSG_delete	ipc(2)
MSG_create	ipc(2)
SEM_owner	ipc(2)
SEM_create	ipc(2)
SEM_delete	ipc(2)
SEM_mode	ipc(2)
SHM_create	ipc(2)

SHM_delete	ipc(2)
SHM_owner	ipc(2)
SHM_mode	ipc(2)
PRIV_userchange	setuid(2), setuid32(2), seteuid(2), seteuid32(2), setreuid(2), setreuid32(2), setresuid(2), setresuid32(2)
PROC_execute	execve(2)
PROC_realuid	setuid(2)
PROC_auditid	ioctl(2)
PROC_loginid	ioctl(2)
PROC_setuserids	setuid(2), setuid32(2), seteuid(2), seteuid32(2), setreuid(2), setreuid32(2), setresuid(2), setresuid32(2)
PROC_realgid	setgid(2), setgid32(2), setgroups(2), setgroups32(2)
PROC_setgroups	setgid(2), setgid32(2), setegid(2), setegid32(2), setregid(2), setregid32(2), setresgid(2), setresgid32(2), setgroups(2), setgroups32(2)
PROC_privilege	capset(2)
SYS_timechange	adjtimex(2), stime(2), settimeofday(2)
ADMIN_amtu	Abstract Machine Test Utility)

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 2.4.x Linux kernel does not either 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 filesystem DAC controls of the audit device file the kernel patch restricts access by verifying if the caller of an I/O control command hast 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 via special 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 admininistrative privilege (CAP_SYS_ADMIN). This functionality is for the benefit of trusted applications that do wish to generate a single 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, for sure, is the kernel space. System calls and network layer actions are handled by the kernel. System calls and netlink operations are all logged after processing by the kernel has been finished (except for: `ioctl(2)`, `execve(2)`, ...). In order to avoid unnecessary data load user applications can send their own, more abstract, information to the kernel. The kernel will add its headers and attributes and send it back to the audit daemon via 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 exceeeds a certain compile-time limit, any processes trying to generate new events will be blocked until there is room in the queue again. The maximum size of the queue is 1024 entries with 8 KB per entry.

3.3.1 Kernel Source

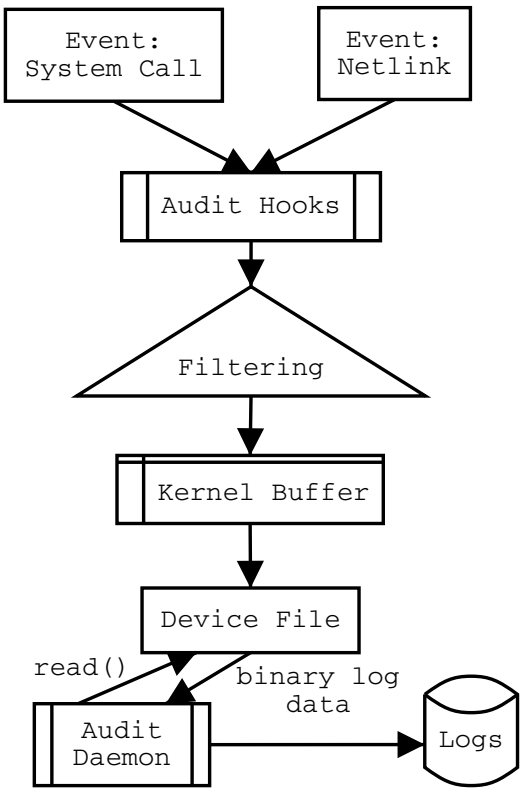

Figure 3.1: Data Flow: Kernel Sources

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

System Calls

As stated before the entry and the exit point of every system call will be monitored. 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).

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 once we are interessted 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

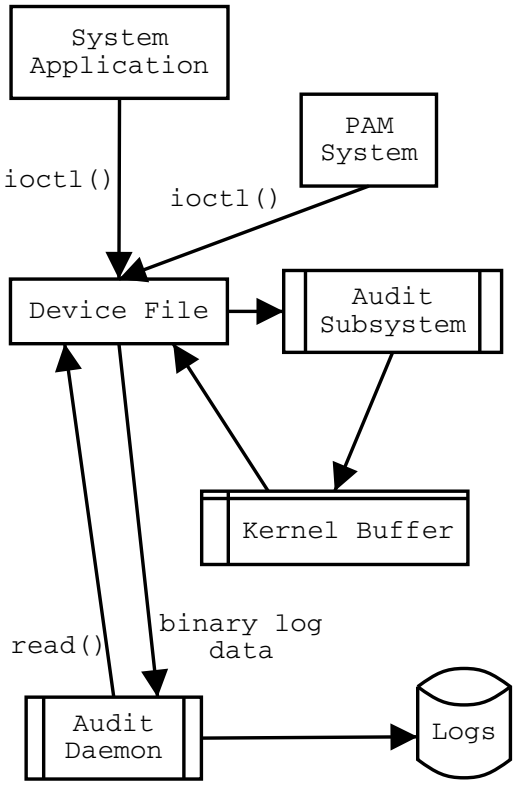

Figure 3.2: Data Flow: User Sources

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“, he 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 a 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 read 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` and `audbin` 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 likes to post- process the log files.

3.9 Who can configure what in which way?

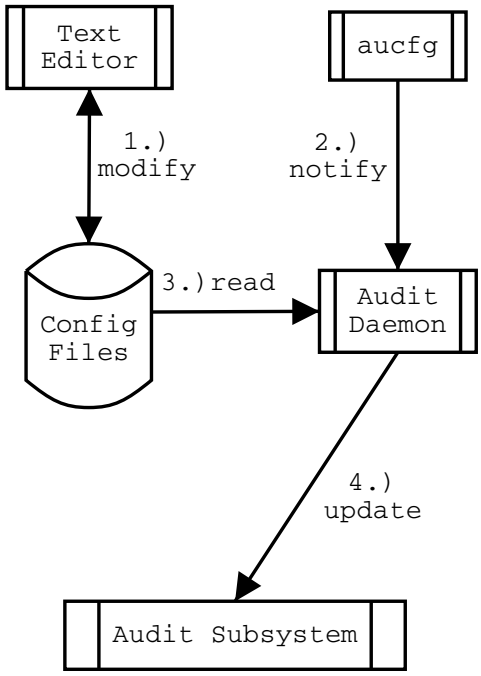

Figure 3.3: Data Flow: Configuration

By using the DAC controls of the filesystem 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 using the tool `aucfg`. `Aucfg` emits a reload message to force re-reading of the configuration. By applying DAC controls only the root user is able to execute `aucfg`, additionally the audit-subsystem only accepts messages generated by user root.

3.10 How is the configuration transfered 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 is a kernel patch to enable system call logging, 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 Lib-PAM, a PAM module, and modified system applications. The following diagram is an overview of the LAuS components:

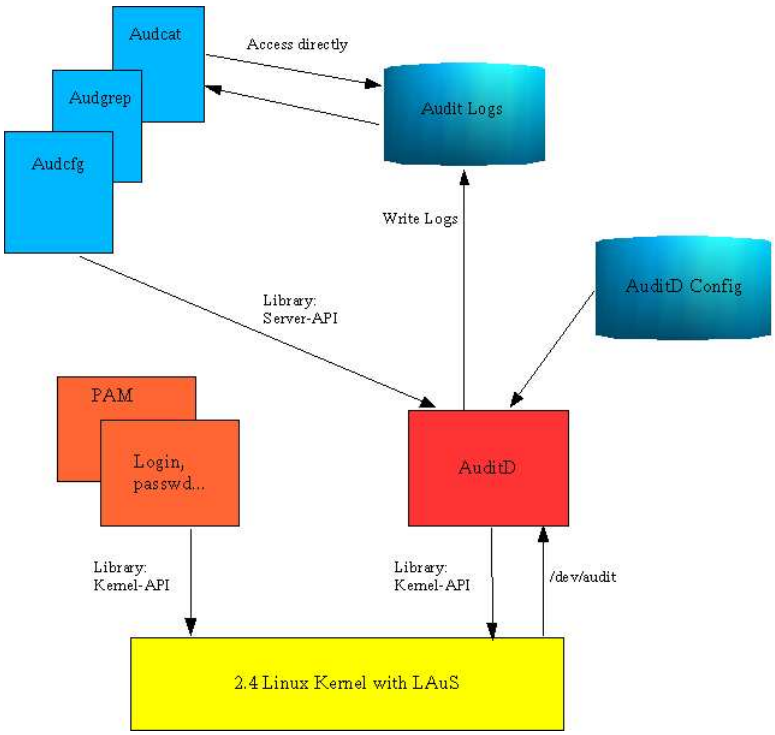

Figure 4.1: LAuS Overview

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 two 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 send to the kernel and includes information like hostname, IP address, terminal name, name of the executeable too. The "Login ID" is stored in the structure `aud_process` and should not be confused with the "Audit ID". The "Login ID" it he "User ID" of the user logged in, and the "Audit ID" is a unique session identifier. Therefore, there can be a session with the same "Login ID" but never with the same "Audit ID".

Audit ID

In addition to the "Login ID", a "Audit ID" is stored in the structure `aud_process` to identify the trail of a process tree. The "Audit ID" is unique and will be assigned to every process attached to the audit-subsystem. If the process spawns a child process this ID gets inherited.

Task Structure

The process task structure as defined in `linux-2.4.19.SuSE/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.

Single Point of Entry and Exit (i386)

To intercept every system call that is made, the kernel patch needs to hook the audit-subsystem into the ptrace framework. These entry points are located in the assembler source file `entry.S` at the jump points `traxesys` and `tracesys_exit`. The following piece of code will describe this method:

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x22,tsk_ptrace(%ebx)    # PT_TRACESYS|PT_AUDITED
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
[...]

tracesys:
        movl $-ENOSYS,EAX(%esp)
        movl %esp,%eax
        pushl %eax
        call SYMBOL_NAME(syscall_trace_enter)
        addl $4,%esp
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae tracesys_exit
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
tracesys_exit:
        movl %esp,%eax
        pushl %eax
        call SYMBOL_NAME(syscall_trace_leave)
        addl $4,%esp
        jmp ret_from_sys_call
```

Here you can see that the assembler code relating to the audit-subsystem in conjunction with ptrace.

If auditing is enabled for a process the code jumps to `tracesys` and executes our intercept function right before the system call is entered. Futhermore the file `ptrace.c` needs adjustment to call the audit functions.

```
asmlinkage void syscall_trace_enter(struct pt_regs *regs)
{
#if defined(CONFIG_AUDIT) || defined(CONFIG_AUDIT_MODULE)
        if (current->ptrace & PT_AUDITED)
                audit_intercept(regs);
#endif

        if ((current->ptrace & (PT_PTRACED|PT_TRACESYS)) == (PT_PTRACED|PT_TRACESYS)
                syscall_ptrace();
}

asmlinkage void syscall_trace_leave(struct pt_regs *regs)
{
#if defined(CONFIG_AUDIT) || defined(CONFIG_AUDIT_MODULE)
        if (current->ptrace & PT_AUDITED)
                audit_result(regs);
#endif

        if ((current->ptrace & (PT_PTRACED|PT_TRACESYS)) == (PT_PTRACED|PT_TRACESYS)
                syscall_ptrace();
}
```

If auditing is enabled while compiling the kernel, either as module or as part of the kernel, and the process has the flag `PT_AUDITED` set, then all necessary informations will be gathered and processed by `audit_intercept` or `audit_result`.

The disadvantage of this mechanism is that every architecture has it's own `entry.S` file and it's own type of CPU registers. So, the changes have to be ported to the other architectures too.

Another problem may occur due to the fact that Linux provides different execution domains for different formats like `a.out`, Solaris executables and so on. To circumvent this problem the EAL3 kernel has to be shipped without the corresponding kernel modules and additionally disabled kernel options.

If other execution level domains are disabled and auditing is enabled every system call has to pass the auditing functions as described above.

Audited System Calls

LAuS catches **every** syscall that is made. But at the moment not all syscall arguments are analyzed. The following table shows all syscalls where LAuS analyses the arguments and all syscalls that are needed for CAPP.

Syscall Name	**needed?**	**analyzed?**
_sysctl	no	yes
access	yes	yes
adjtimex	no	yes
brk	yes	yes
capset	yes	yes
chdir	yes	yes
chmod	yes	yes
chown	yes	yes
chown32	yes	yes
chroot	no	yes
clone	no	yes
close	no	yes
create	yes	yes
create_module	yes	yes
delete_module	yes	yes
execve	yes	yes
exit	no	yes
fchdir	no	yes
fchmod	yes	yes
fchown	yes	yes
fchown32	yes	yes
fgetxattr	no	yes
flistxattr	no	yes
fork	no	yes
fremovexattr	yes	yes
fsetxattr	yes	yes
ftruncate	no	yes
ftruncate64	no	yes
getxattr	no	yes
init_module	yes	yes
ioperm	yes	yes
iopl	yes	yes
ipc (msgctl, msgget, semctl, semget, shmat, shmctl, shmget)	yes	yes

kill	no	yes
lchown	yes	yes
lchown32	yes	yes
lgetxattr	no	yes
link	yes	yes
listxattr	no	yes
llistxattr	no	yes
lremovexattr	yes	yes
lsetxattr	yes	yes
mkdir	yes	yes
mknod	yes	yes
mount	yes	yes
open	yes	yes
ptrace	yes	yes
query_module	no	yes
read	no	yes
reboot	no	yes
removexattr	yes	yes
rename	yes	yes
rmdir	yes	yes
sched_setaffinity	no	yes
sched_setparam	no	yes
sched_setscheduler	no	yes
setdomainname	no	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
sethostname	no	yes
setpriority	no	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
setrlimit	no	yes
settimeofday	no	yes
setuid	yes	yes
setuid32	yes	yes
setxattr	yes	yes
socketcall (bind)	yes	yes
stime	no	yes
swapoff	no	yes
swapon	yes	yes
symlink	yes	yes
syslog	no	yes
tkill	no	yes
truncate	yes	yes
truncate64	yes	yes
umask	yes	yes
umount	no	yes
umount2	no	yes
unlink	yes	yes
uselib	no	yes
utime	yes	yes
vfork	no	yes
write	no	yes

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 list of I/O control messages for which data is included in the event is (network):

SIOCADDMULTI:	Multicast address lists
SIOCADDDLCI:	Create new DLCI device
SIOCADDRT:	add routing table entry
SIOCCHGTUNNEL	
SIOCDELTUNNEL	
SIOCADDTUNNEL	
SIOCDARP	
SIOCDELRT:	delete routing table entry

SIOCETHTOOL:	
SIOCDELMULTI.	
SIOCDIFADDR:	delete PA address
SIOCDARP:	delete ARP table entry
SIOCDELDLCI:	Delete DLCI device
SIOCGIFBR:	
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
SIOCSIFBR	
SIOCSIFFLAGS:	set flags
SIOCSIFHWADDR:	
SIOCSIFLINK:	set iface channel
SIOCSIFTXQLEN:	
SIOCSMIIREG:	
SIOCSIFHWADDR.	set hardware address
SIOCSIFENCAP:	
SIOCSIFSLAVE:	
SIOCSIFPFLAGS.	set/get extended flags set
SIOCSIFHWBROADCAST:	set hardware broadcast addr
SIOCSIFBR:	Set bridging options
SIOCSIFTXQLEN:	Set the tx queue length
SIOCSARP:	set ARP table entry
SIOCSIFMAP:	Set device parameters

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 is 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 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 named /dev/audit and has the major number 10 (misc devices) and minor number 224. **Note:** Namespace-collisions can happen with block device /dev/audit, major number 103.

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	
AUIOCUSERMESSAGE	struct audit_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 Kernel API Library

```
Author: Thomas Biege <thomas@suse.de>
Date: 2003-06-17
Version: 0.4
Todo: - add filter-functions
------------------------------------------------------------------------


Kernel-API Library Functions
============================

definition:     int laus_init(void)
return value:   < 0: error
error codes:    none
arguments:      none
used by:        audit daemon, audit tools, system applications
description:    Initialise runtime parameters


definition:     int laus_open(char *dev_file)
return value:   < 0: error, file descriptor
error codes:    LERR_OPEN_FAILED
arguments:      device-file of systrace, if NULL a default value
                will be used
used by:        audit daemon, audit tools, system applications
description:    Opens systrace-interface.


definition:     int laus_log(const char *fmt, ...)
return value:   < 0: error
arguments:      printf-style format-string
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
description:    Write printf-style message to kernel. The kernel will fill
                in the regular headers and forward the message to the audit-
                daemon


definition:     int laus_registerauditid(void)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
```

```
arguments:      none
used by:        audit daemon
description:    Register the current process as audit-daemon to
                the kernel.


definition:     pid_t laus_exec(int flags, char *prog_name, ...)
return value:   < 0: error, process-id
error codes:    LERR_FORK_FAILED
                LERR_OUT_OF_MEMORY
                LERR_EXEC_FAILED,
                LERR_PROCESS_CRASHED
arguments:      file descriptor, execution flags (NONE, DETACH), program-
                name, optional program-arguments
used by:        audit tools (i.e. wrapper for other daemons), system
                applications
description:    Fork and execute a program.


definition:     int laus_attach(void)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
arguments:      none
used by:        audit tools, system applications
description:    Attach the current process.


definition:     int laus_setauditid(void)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
arguments:      none
used by:        audit tools, system applications
description:    Set the audit-id for the current process. The audit-id
                is needed to keep track of user activities even if they
                change their user/group-id.


definition:     int laus_setsession(id_t uid, const char *hostname,
                const char *address, const char *terminal)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
```

```
                LERR_IOCTL_FAILED
arguments:      user-id, remote hostname, remote ip-address, terminal
used by:        audit tools, system applications
description:    Set the terminal-id for the current process. The terminal-id is
                needed to keep track of the "line" a user chooses to enter the
                system. (Also refered to as "audit login message")


definition:     int laus_clrpolicy(void)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
arguments:      none
used by:        audit daemon
description:    Clear policies in kernel-space.


definition:     int laus_setpolicy(int syscall, int action, int filter)
return value:   < 0: error
error codes:    LERR_NOT_OPENED
                LERR_IOCTL_FAILED
arguments:      syscall to audit, action, filter to apply
used by:        audit daemon
description:    Assign an action to a syscall. Asigning a filter is not
                implemented yet.


definition:     int laus_read(void *buffer, size_t size)
return value:   < 0: error, > 0: number of bytes read, = 0: EOF
error codes:    LERR_NOT_OPENED
                LERR_READ_FAILED
arguments:      pointer to a buffer, buffer size
used by:        audit daemon
description:    Read output from the kernel.


definition:     const char * laus_strerror(int code)
return value:   error string
error codes:    none
arguments:      return value of laus-functions
used by:        audit daemon, audit tools, system applications
description:    This function can be used to translate numerical
                error-codes into a more descriptive error-string.
```

```
definition:     int laus_detach(void)
return value:   < 0: error
error codes:    none
arguments:      none
used by:        audit tools, system applications
description:    Detach the running process.


definition:     int laus_close(int fd)
return value:   < 0: error
arguments:      file descriptor
used by:        audit daemon, audit tools, system applications
description:    Close systrace-interface.



Kernel-API Library Structures
=============================

struct aud_message {
  u_int32_t     msg_seqnr;
  u_int16_t     msg_type;
  u_int16_t     msg_arch;

  pid_t         msg_pid;
  size_t        msg_size;

  unsigned long   msg_timestamp;
  unsigned int    msg_audit_id;
  unsigned int    msg_login_uid;
  unsigned int    msg_euid, msg_ruid, msg_suid, msg_fsuid;
  unsigned int    msg_egid, msg_rgid, msg_sgid, msg_fsgid;

  union {
    char        dummy;
  } msg_data;
};

struct aud_msg_child {
  pid_t new_pid;
};
```

```
struct aud_msg_syscall {
  int           personality;

  /* System call codes can have major/minor number.
   * for instance in the socketcall() case, major
   * would be __NR_socketcall, and minor would be
   * SYS_ACCEPT (or whatever the specific call is).
   */
  int           major, minor;

  int           result;
  unsigned int  length;
  unsigned char data[1];  /* variable size */
};

struct aud_msg_netlink {
  unsigned int  groups, dst_groups;
  int           result;
  unsigned int  length;
  unsigned char data[1];  /* variable size */
};

struct aud_msg_login {
  uid_t  uid;
  char   hostname[AUD_MAX_HOSTNAME];
  char   address[AUD_MAX_ADDRESS];
  char   terminal[AUD_MAX_TERMINAL];
  char   executable[PATH_MAX];
};
```

For more information see man page `laus_record(7)`.

4.1.3 Server API Library

```
Author: Thomas Biege <thomas@suse.de>
Date: 2003-06-18
Version: 0.3 (Changes are very likely)
Todo:
------------------------------------------------------------------------------


Server-API Library Functions
============================
```

```
Parsing:
--------
definition:     int laussrv_process_log(const char *filename,
                audit_callback_fn_t *func)
return value:   < 0: error
arguments:      name of log-file, callback function.
used by:        system applications
description:    Read audit-logs and call the callback function to handle the
                data.


Controlling:
------------
definition:     void laussrv_ctrl_open(void)
return value:   < 0: error
arguments:      none
used by:        audit tools
description:    This function opens the control-channel (unix domain
                socket) to the audit-server. It enables the LAuS
                command-line tools to control the Linux-Auditsubsystem
                in a predefined and less error-prone way.

definition:     int laussrv_ctrl(struct ctrl_message *msg)
return value:   < 0: error
arguments:      control-message pointer
used by:        audit tools
description:    Send control-message to audit-server.

definition:     int laussrv_ctrl_setpolicy(int fd, pid_t pid, char *policy)
return value:   < 0: error, policy-id
arguments:      none, process-id, policy
used by:        audit tools
description:    Set policy for a given process.

definition:     int laussrv_ctrl_getpolicy(int fd, int policy_id)
return value:   < 0: error
arguments:      none, policy-id
used by:        audit tools
description:    Get policy for a given policy-id.

definition:     int laussrv_ctrl_delpolicy(int fd, int policy_id)
return value:   < 0: error
```

```
arguments:      none, policy-id
used by:        audit tools
description:    Delete policy belonging to policy_id.

definition:     int laussrv_ctrl_close(int fd)
return value:   < 0: error
arguments:      none
used by:        audit tools
description:    close connection to audit-server

definition:     int audit_print(time_t timestamp, struct aud_message *msg,
                int flags)
return value:   0
arguments:      the arguments are set via the callback function by
                laussrv_process_log()
used by:        audit tools
description:    Output audit data in human readable format.
```

```
Server-API Library Structures
=============================

Parsing:
--------
typedef int audit_callback_fn_t(time_t timestamp,
                                struct aud_message *msg,
                                struct aud_message *related,
                                int flags);


Controlling:
------------
#define LAUS_TYPE_GO
#define LAUS_TYPE_HALT
#define LAUS_TYPE_RELOAD
#define LAUS_TYPE_GET_STATUS
#define LAUS_TYPE_DBG_INC
#define LAUS_TYPE_DBG_DEC

struct ctrl_message
{
```

```
        int   version_major, version_minor;
        int   endian;

        u_long        type;
        union
        {
                struct status        state;

        } data;
}

struct status
{
        pid_t              pid;
        pid_t              ppid
        boolean            traced;
        int                audit_id;
        int                audit_session_id
        tid_t              terminal_id;
}
```

4.1.4 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.

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

In bin mode, four 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 and then the secondary file is utilized until it reaches its maximum capacity at which point the first file is used again. This allows the administrator to specify the maximum disk space that audit records will ever take.

Note: The default configuration uses `/var/log/audit` as append-file and also as symlink to the current bin-file in bin-mode. If you use append-mode and switch to bin-mode your audit data in `/var/log/audit` gets lost! Please backup your data before switching modes or change your configuration.

The following command–line options are recognized:

–r Reload the system call filters in the kernel with out interrupting collection of audit events. This is better than restarting the daemon, because no audit events will be lost.

–F Run in foreground, and log all error diagnostics and debug messages to standard error rather than to syslog.

–d Enable debugging messages. Specifying this option repeatedly will increase verbosity

4.1.5 Audit Tools

The user space tools consist of `aucat`, `augrep`, `aurun`, `aucfg`, and `audbin`. `Aucat` reads the audit log files and outputs the records in human readable format. The administrator can select between ASCII, SQL and IDMEF [1] 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. `aucfg` provides the interface that allows the administrator to inform the audit daemon of changes to the configuration and to start and stop auditing. `Aurun` can be used as a wrapper to start applications, like `Apache`, and attach them to the audit-subsystem without modifying the applications source code. `Audbin` is for post-processing bin-files.

aucat

To read and post-process the audit logs `aucat` can be used wih the following options:

–f Process audit records read from FILENAME. Default is “/var/log/audit“.

–?	Print out help screen.
–h, —header	Print out a header at the top of output that identifies the columns of the output.
-t- TIMEFORMAT	Change format of time that is output. Default is “iso8601“. Options are: **iso8601:** print time in iso 8601 format. (YYYY-MM-DDT hh:mm:ss) **unix:** print time in (DD MM YY hh:mm:ss). **raw:** print raw time. **none:** do not print time.
–v	Print out all variables in message, not all are printed by default.

augrep

`Augrep` can be used to search by using various attributes. The output can be formated. The following options are supported.

–f	Process audit records read from FILENAME. Default is “/var/log/audit“.
–?	Print out help screen.
–h, —header	Print out a header at the top of output that identifies the columns of the output.
–t TIMEFORMAT	Change format of time that is output. Default is “iso8601“. Options are: **iso8601:** print time in iso 8601 format. (YYYY-MM-DDT hh:mm:ss) **unix:** print time in (DD MM YY hh:mm:ss). **raw:** print raw time. **none:** do not print time.
–v, —verboseall	Print out all variables in message, not all are printed by default.
–a SESSION_ID, —auditid=SESSION_ID	Find audit record(s) with specified session id.
–l LOGIN_NAME, —loginid=LOGIN_NAME	Find audit record(s) with specified login id. (NOTE: This option cannot be used if option “uid“ has already been specfied.)
–n SEQ_NUM, —sequencenum=SEQ_NUM	Find audit record(s) with specified sequence number.
–p PID, —pid=PID	Find audit record(s) with specified pid.
–u UID, —uid=UID	Find audit record(s) with specified uid.

	(NOTE: This option cannot be used if option “loginid“has already been specfied.)
—euid=EUID	Find audit record(s) with specified euid.
—egid=EGID	Find audit record(s) with specified egid.
—fsuid=FSUID	Find audit record(s) with specified fsuid.
—fsgid=FSGID	Find audit record(s) with specified fsgid.
—ruid=RUID	Find audit record(s) with specified ruid.
—rgid=RGID	Find audit record(s) with specified rgid.
—suid=SUID	Find audit record(s) with specified suid.
—sgid=SGID	Find audit record(s) with specified sgid.
–s STARTT , —starttime=STARTT	Find audit record(s) that started at or after a specified start time. (Note: Time must be in iso8601 Format “YYYY-MM-DDThh:mm:ss“)
–s ENDT, —endtime=ENDT	(Note: Time must be in iso8601 Format “YYYY-MM-DDThh:mm:ss“)
–x EVENT, —event=EVENT	Find audit record(s) with specified event type. Options: **LOGIN:** Find login messages. **NETLINK:** Find netlink messages. **SYSCALL:** Find syscall messages. **TEXT:** Find messages that come from userspace tools (ex. cron and at)
–A ADDRESS, —address=ADDRESS	Find LOGIN message(s)with specified address.
–E EXECUTE, —execute=EXECUTE	Find LOGIN message(s) with specified executable.
–H HOSTNAME, —hostname=HOSTNAME	Find LOGIN message(s) with specified hostname.
–T TERMINAL, —terminal=TERMINAL	Find LOGIN message(s) with specified terminal.
–G GROUP, —group=GROUP	Find NETLINK message(s) with specified group.
–I GROUP, —dstgroup=DSTGROUP	Find NETLINK message(s) with specified dstgroup.
–L RESULT, —netresult=RESULT	Find NETLINK message(s) with specified result.
–K DATA, —netdata=DATA	Find NETLINK message(s) that contain specified DATA.
–X DATA,	

—textdata=DATA	Find TEXT message(s) that contain DATA.
–S name, —syscall name	Find system call messages matching the given name. This also covers calls such as accept and listen, which are multiplexed through socketcall on some architectures.
–M MAJOR_NUMBER, —major=MAJOR_NUMBER	Find SYSCALL message(s) with specified major number.
–N MINOR_NUMBER, —minor=MAJOR_NUMBER	Find SYSCALL message(s) with specified minor number.
–R RESULT, —sysresult=RESULT	Find SYSCALL message(s) with specified result.
–D DATA, —sysdata=DATA	Find SYSCALL message(s) that contain specified DATA.

aucfg

To control the audit daemon via the command line `aucfg` can be used. Controlling the audit daemon is done by using the following options:

—reload:	Reload configuration
—suspend:	Suspend auditing, but keep configuration
—resume:	Resume auditing
—dbg_inc:	More verbose debugging
—dbg_dec:	Less verbose debugging

aurun

To attach an application like `dhcpd`, `apache` or alike to the audit-subsystem, `aurun` can be used as a wrapper. The following option are recognized:

–u user	By default, the process will be run with the privilege of the current user. By specifying the -u command line switch, you can specify the name of a user account, the privileges of which the program will be executed with.
–N	Do not write a login session record, use this for programs such as FTP servers that do their own PAM authentication.

Trailing arguments will be used as arguments for the wrapped applications.

audbin

`Audbin`‘s purpose is post-processing and managing of log-files. The following option are recognized:

–S file	Copies the log file to the given destination. The destination can contain the following substituion strings: %u: generate a number to make the file name unique. %t: include the current time stamp as integer %h: include the hostname as given in the header of the original log file. %%: include a verbatim percent character. The special filename “-“ indicates standard output.
–C	Clear the log file after saving its contents. This option can also be used without the -S option.
–o	If the destination file exists, overwrite it.
–a	If the destionation file exists, append the contents of the log file to it.
–q	Do not print any diagnostic messages to standard output

The last option given must be the file containing the audit–data.

4.1.6 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 `pam_laus` module 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` or `session` modules. An Audit User Message is written to the audit log indicating success or failure as determined by the module stack's returned value. An Audit Login Message is also written using `laus_setsession()` if no session data is currently associated with the process.

The PAM configuration for each service MUST ensure that the `pam_laus` 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` module in that path will not work.

4.1.7 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.

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 threshold and alike. The files `filter.conf` and `filesets.conf` (not mandatory, just used to ease configuration) are used for filtering.

auditd.conf

The following just shows an example config file:

```
# kernel interface
device-file = "/dev/audit";

# filter config
filter = "/etc/audit/filter.conf";

output {
        mode            = append;              # append to log
        file-name       = "/var/log/audit";
};

# Alternative output
# output {
#       mode            = stream;
#       command         = "/usr/local/sbin/send_to_syslog"
# }

# Another output alternative:
# output {
#       mode            = bin;
#       num-files       = 4;
#       file-size       = 20M;
#       file-name       = "/var/log/audit.d/bin";
#       notify          = "/usr/local/sbin/audbin-notify";
# }

# threshold for running out of disc space
threshold disk-space-low {
        space-left = 10M;
        action {
                type = syslog
                facility = security;
                priority = warning;
        };
        action {
                type = notify
                command = "/usr/local/bin/page-admin";
        };
};

threshold disk-full {
        space-left = 0;
        action {
```

```
                type = shutdown
                # no options
        };
};
```

The audit daemon is able to handle more then one output section simultaneously. As you can see, the system will be shutdown when the disk runs out of space.

filesets.conf

The following just shows an example config file:

```
#
# This file contains file name sets etc used in the default
# audit filter configuration file.
#
# The syntax of this file is described in filter.conf(5).
#

#
# Set of files for which we track read access.
#
set secret-files = {
"/etc/shadow",
"/etc/gshadow",
"/var/log/audit",
"/var/log/audit.d",
"/var/log/audit.d/bin.0",
"/var/log/audit.d/bin.1",
"/var/log/audit.d/bin.2",
"/var/log/audit.d/bin.3",
};
```

filter.conf

The following section shows an example configuration for the audit filter. It includes enuogh comments and covers a wide range of cases and is an excellent starting point for writing futher filter rules. **Note:** The glibc-API is different from the kernel-API. Using `setreuid(2)` in a program doesn't necessarily trigger the `setreuid(2)` system-call, instead `setreuid32(2)` reveals. This is due to the fact that the glibc wrappes architecture-specific behaviour.

```
#
# This is a sample filter.conf file.
```

```
# Please take a look at filesets.conf first if you
# wish to customize what system calls will be logged.
#
# The syntax of this file is described in filter.conf(5).
#

#
# Various primitive predicates
predicate is-null = eq(0);
predicate is-negative = lt(0);
predicate is-system-uid = lt(100);

#
# Predicate to check open(2) mode: true iff
# (mode & O_ACCMODE) == O_RDONLY
predicate is-rdonly = mask(O_ACCMODE, O_RDONLY);

#
# Predicates for testing file type, valid when applied
# to a file type argument
predicate __isreg = mask(S_IFMT, S_IFREG);
predicate __isdir = mask(S_IFMT, S_IFDIR);
predicate __ischr = mask(S_IFMT, S_IFCHR);
predicate __isblk = mask(S_IFMT, S_IFBLK);
predicate __issock = mask(S_IFMT, S_IFSOCK);
predicate __islnk = mask(S_IFMT, S_IFLNK);
predicate s_isreg = __isreg(file-mode);
predicate s_isdir = __isdir(file-mode);
predicate s_ischr = __ischr(file-mode);
predicate s_isblk = __isblk(file-mode);
predicate s_issock = __issock(file-mode);
predicate s_islnk = __islnk(file-mode);
predicate is-tempdir = mask(01777, 01777);
predicate is-world-writable = mask(0666, 0666);

#
# Predicates dealing with process exit code
predicate if-crash-signal =
!mask(__WSIGMASK, 0)
 && (mask(__WSIGMASK, __WSIGILL) ||
     mask(__WSIGMASK, __WSIGABRT) ||
     mask(__WSIGMASK, __WSIGSEGV) ||
     mask(__WSIGMASK, __WSIGSTKFLT));
```

```
#
# Predicates for audit-tags
predicate is-o-creat = mask(O_CREAT, O_CREAT);
predicate is-ipc-remove = eq(IPC_RMID);
predicate is-ipc-setperms = eq(IPC_SET);
predicate is-ipc-creat = mask(IPC_CREAT, IPC_CREAT);
predicate is-auditdevice = prefix("/dev/audit");
predicate is-cmd-set-auditid = eq(AUIOCSETAUDITID);
predicate is-cmd-set-loginid = eq(AUIOCLOGIN);

#
# Misc filters
filter is-root = is-null(uid);
filter is-setuid = is-null(dumpable);
filter syscall-failed = is-negative(result);
predicate is-af-packet = eq(AF_PACKET);
predicate is-af-netlink = eq(AF_NETLINK);
predicate is-sock-raw = eq(SOCK_RAW);

#
# Include filesets.
#
include "filesets.conf";


#
# "Secret" files should not be read by everyone -
# we also log read access to these files
#
predicate is-secret = prefix(@secret-files);

#
# All regular files owned by a system uid are deemed sensitive
#
predicate is-system-file = is-system-uid(file-uid)
      && !prefix("/var")
      && !is-world-writable(file-mode);

#
# Define ioctls we track
#
set sysconf-ioctls = {
```

```
SIOCADDDLCI,
SIOCADDMULTI,
SIOCADDRT,
SIOCBONDCHANGEACTIVE,
SIOCBONDENSLAVE,
SIOCBONDRELEASE,
SIOCBONDSETHWADDR,
SIOCDARP,
SIOCDELDLCI,
SIOCDELMULTI,
SIOCDELRT,
SIOCDIFADDR,
SIOCDRARP,
SIOCETHTOOL,
SIOCGIFBR,
SIOCSARP,
SIOCSIFADDR,
SIOCSIFBR,
SIOCSIFBRDADDR,
SIOCSIFDSTADDR,
SIOCSIFENCAP,
SIOCSIFFLAGS,
SIOCSIFHWADDR,
SIOCSIFHWBROADCAST,
SIOCSIFLINK,
SIOCSIFMAP,
SIOCSIFMEM,
SIOCSIFMETRIC,
SIOCSIFMTU,
SIOCSIFNAME,
SIOCSIFNETMASK,
SIOCSIFPFLAGS,
SIOCSIFSLAVE,
SIOCSIFTXQLEN,
SIOCSMIIREG
};
predicate is-sysconf-ioctl = eq(@sysconf-ioctls);

#
# System calls on file names
#
set file-ops = {
"mkdir", "rmdir", "unlink",
```

```
"chmod",
        "chown", "lchown",
"chown32", "lchown32",
};

#
# General system related ops
#
set system-ops = {
swapon, swapoff,
create_module, init_module, delete_module,
sethostname, setdomainname,
};

set priv-ops = {
"setuid",
"setuid32",
"seteuid",
"seteuid32",
"setreuid",
"setreuid32",
"setresuid",
"setresuid32",
"setgid",
"setgid32",
"setegid",
"setegid32",
"setregid",
"setregid32",
"setresgid",
"setresgid32",
"setgroups",
"setgroups32",
"capset",
};

#
# Audit-Tags (only syscall related tags are handled here)
#

# define sets of syscalls related to audit-tags

# System calls for changing file modes
```

```
set mode-ops = {
"chmod",
"fchmod",
};

# System calls for changing file owner
set owner-ops = {
"chown", "lchown",
"chown32", "lchown32",
"fchown",
};


# System calls doing file link operations
set link-ops = {
"link", "symlink",
};

# System calls for creating device files
set mknod-ops = {
"mknod",
};

# System calls for opening a file
set open-ops = {
"open",
};

# File renaming
set rename-ops = {
"rename",
};

# File truncation
set truncate-ops = {
"truncate", "truncate64",
"ftruncate", "ftruncate64",
};

# Unlink files
set unlink-ops = {
"unlink",
};
```

```
# Deletion of directories
set rmdir-ops = {
"rmdir",
};

# Mounting of filesystems
set mount-ops = {
"mount",
};

# Unounting of filesystems
set umount-ops = {
"umount",
"umount2"
};

# Changing user (-role)
set userchange-ops = {
"setuid",
"setuid32",
"seteuid",
"seteuid32",
"setreuid",
"setreuid32",
"setresuid",
"setresuid32",
};

# Execute another program
set execute-ops = {
"execve",
};

# Set real user-ID
set realuid-ops = {
"setuid",
"setuid32",
};

# Set user-IDS in gerneral
set setuserids-ops = {
"setuid",
```

```
"setuid32",
"seteuid",
"seteuid32",
"setreuid",
"setreuid32",
"setresuid",
"setresuid32",
};

# Set real group-ID
set realgid-ops = {
"setgid",
"setgid32",
"setgroups",
"setgroups32",
};

# Set group-IDs in gerneral
set setgroups-ops = {
"setgid",
"setgid32",
"setegid",
"setegid32",
"setregid",
"setregid32",
"setresgid",
"setresgid32",
"setgroups",
"setgroups32",
};

# Set other kind of privileges (capabilities)
set privilege-ops = {
"capset",
};

# Change system-time
set timechange-ops = {
"adjtimex",
"stime",
"settimeofday",
};
```

```
# bring sets and tags in conjunction

tag "FILE_mode"
syscall @mode-ops = always;

tag "FILE_owner"
syscall @owner-ops = always;

tag "FILE_link"
syscall @link-ops = always;

tag "FILE_mknod"
syscall @mknod-ops = always;

tag "FILE_create"
syscall open = is-o-creat(arg1);
tag "FILE_create"
syscall creat = always;

#tag "FILE_open"
#syscall @open-ops = always;

tag "FILE_open"
syscall @open-ops = (is-system-file(arg0) && !(is-rdonly(arg1)))
    || is-secret(arg0);

tag "FILE_rename"
syscall @rename-ops = always;

tag "FILE_truncate"
syscall @truncate-ops = always;

tag "FILE_unlink"
syscall @unlink-ops = always;

tag "FS_rmdir"
syscall @rmdir-ops = always;

tag "FS_mount"
syscall @mount-ops = always;

tag "FS_umount"
```

```
syscall @umount-ops = always;

# I think owner changing doesnt make much sense
tag "MSG_owner"
syscall msgctl = is-ipc-setperms(arg1);

tag "MSG_mode"
syscall msgctl = is-ipc-setperms(arg1);

tag "MSG_delete"
syscall msgctl = is-ipc-remove(arg1);

tag "MSG_create"
syscall msgget = always;

tag "SEM_owner"
syscall semctl = is-ipc-setperms(arg2);

tag "SEM_mode"
syscall semctl = is-ipc-setperms(arg2);

tag "SEM_delete"
syscall semctl = is-ipc-remove(arg2);

tag "SEM_create"
syscall semget = always;

tag "SHM_owner"
syscall shmctl = is-ipc-setperms(arg1);

tag "SHM_mode"
syscall shmctl = is-ipc-setperms(arg1);

tag "SHM_delete"
syscall shmctl = is-ipc-remove(arg1);

tag "SHM_create"
syscall shmget = always;

tag "PRIV_userchange"
syscall @userchange-ops = always;

tag "PROC_execute"
```

```
syscall @execute-ops = always;

tag "PROC_realuid"
syscall @realuid-ops = always;

tag "PROC_auditid"
syscall ioctl = (is-auditdevice(arg0) && is-cmd-set-auditid(arg1));

tag "PROC_loginid"
syscall ioctl = (is-auditdevice(arg0) && is-cmd-set-loginid(arg1));

tag "PROC_setuserids"
syscall @setuserids-ops = always;

tag "PROC_realgid"
syscall @realgid-ops = always;

tag "PROC_setgroups"
syscall @setgroups-ops = always;

tag "PROC_privilege"
syscall @privilege-ops = always;

tag "SYS_timechange"
syscall @timechange-ops = always


# not required by CAPP
syscall ipc = always;

syscall socket = is-af-packet(arg0) || is-sock-raw(arg1);
syscall ioctl = is-sysconf-ioctl(arg1);

#
# Special filters for process/termination
event process-exit = if-crash-signal(exitcode);

#
# Events we want to log unconditionally:
event network-config = always;
event user-message = always;
event process-login = always;
```

4.3 LAuS Log Files

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

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. An audit record is constructed by using the following data structures:

```
struct laus_record_header {
  time_t  r_time;
  size_t  r_size;
}

struct aud_message {
  u_int32_t     msg_seqnr;
  u_int16_t     msg_type;
  u_int16_t     msg_arch;

  pid_t         msg_pid;
  size_t        msg_size;

  unsigned long   msg_timestamp;
  unsigned int    msg_audit_id;
  unsigned int    msg_login_uid;
  unsigned int    msg_euid, msg_ruid, msg_suid, msg_fsuid;
  unsigned int    msg_egid, msg_rgid, msg_sgid, msg_fsgid;

  union {
    char        dummy;
  } msg_data;
};
```

The following structures are optional and depend on the message type. They can be accessed by using the `dummy` variable of `struct aud_message`.

```
struct aud_msg_child {
  pid_t new_pid;
};

struct aud_msg_syscall {
```

```
  int           personality;

  /* System call codes can have major/minor number.
   * for instance in the socketcall() case, major
   * would be __NR_socketcall, and minor would be
   * SYS_ACCEPT (or whatever the specific call is).
   */
  int           major, minor;

  int           result;
  unsigned int  length;
  unsigned char data[1];  /* variable size */
};

struct aud_msg_netlink {
  unsigned int  groups, dst_groups;
  int           result;
  unsigned int  length;
  unsigned char data[1];  /* variable size */
};
```

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:

```
2003-08-21T04:30:00     31   1060     root [PROC_execute] execve("/bin/rm",
                                           [data, len=0], [data, len=0])
2003-08-21T04:30:00     32   1060     root [FILE_unlink] unlink("/tmp/run-crons
                                           .zxPaul"); result=-21 ["Is a directory"]
2003-08-21T04:30:00     33   1060     root [FS_rmdir] rmdir("/tmp/run-crons.
                                            zxPaul"); result=0
2003-08-21T04:42:30     34   1061       -1 [AUTH_success] pam: Accounting
                                           succeeded for user=thomas (hostname=
```

```
                                                ras.suse.de, addr=10.0.8.6, terminal
                                                =NODEVssh)
2003-08-21T04:44:07     35   1099       -1 [AUTH_success] pam: Authentication
                                           succeeded for user=root (hostname=?,
                                           addr=?, terminal=pts/0)
2003-08-21T04:44:07     36   1099       -1 [AUTH_success] pam: Accounting
                                           succeeded for user=root (hostname=?,
                                           addr=?, terminal=pts/0)
2003-08-21T04:44:07     37   1099       -1 [AUTH_success] pam: Session open
                                           succeeded for user=root (hostname=?,
                                           addr=?, terminal=pts/0)
2003-08-21T04:45:00     38   1136       -1 cron: executing cron job - crontab=
                                           /etc/crontab, uid=0, gid=0, cmd=
                                           test -x /usr/lib/cron/run-crons &&
                                           /usr/lib/cron/run-crons >/dev/null
                                           2>&1
2003-08-21T04:45:00     39   1136       -1 [PROC_auditid] ioctl("/dev/audit",
                                           AUIOCSETAUDITID, [data, len=0]);
                                           result=0
2003-08-21T04:45:00     40   1136     root LOGIN: uid=0, terminal=cron job,
                                           executable=/usr/sbin/cron
2003-08-21T04:45:00     41   1136     root [PROC_loginid] ioctl("/dev/audit",
                                            AUIOCLOGIN, [data, len=0]);
                                           result=0
2003-08-21T04:45:00     42   1136     root [PROC_execute] execve("/bin/bash",
                                           [data, len=0], [data, len=0])
2003-08-21T04:45:00     43   1136     root [FILE_open] open("/dev/tty", O_RDWR
                                           |O_NONBLOCK|O_LARGEFILE, 0); result=
                                           -6 ["No such device or address"]
2003-08-21T04:45:00     62   1152     root [FILE_unlink] unlink("/tmp/run-crons
                                           .PxjwME"); result=-21 ["Is a directory"]
2003-08-21T04:45:00     63   1152     root [FS_rmdir] rmdir("/tmp/run-crons
                                           .PxjwME"); result=0
2003-08-21T04:51:48     64   1362       -1 useradd: user added - user=ntp, uid=
                                           74, gid=65534, home=/var/lib/ntp,
                                           shell=/bin/false, by=0
```

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.

Chapter 5

Open Issues

- proofreading by Olaf
- proofreading by Andreas and Helmut

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

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