Praktikumsbericht

Betriebssystem-Audit für Linux

Erweiterung des freiverfügbaren Betriebssystem Linux, um ein Betriebssystem-Audit

Oliver Stecklina <stecklina@secunet.de>

9. Oktober 2001
Version 1.0.0

Mentor: Birk Richter <b.richter@secunet.de>

Betreuer: Mario Zühlke <mz@informatik.tu-cottbus.de>

Inhaltsverzeichnis

Kapitel 1 **Einleitung** **1**

Kapitel 2 **Bestandsaufnahme von bestehenden Betriebssystem-Audit-Konzepten** **4**
2.1 Audit-Konzepte in marktgängigen Betriebssystemen 4
2.1.1 Das *Basic Security Modul (BSM)* von Solaris 4
2.1.1.1 Bestandteile des BSM von Solaris 4
2.1.1.2 Funktionsumfang des BSM 5
2.1.2 Ereignis-Protokollierung unter Windows NT 7
2.1.2.1 Bestandteile des Windows NT Audit 7
2.1.2.2 Funktionsumfang des Windows NT Audit 8
2.2 Bestehende Konzepte bzw. Ansätze für Linux 9
2.2.1 Der Syslog-Mechanismus 9
2.2.2 Der Audit-Dämon vom H.E.R.T. 10
2.3 Vergleich der Audit-Funktionen 11
2.3.1 Aufzeichnungsmöglichkeiten 11
2.3.2 Filter- bzw. Konfigurationsmöglichkeiten 12

Kapitel 3 **LOSA – Linux Operating System Audit** **14**
3.1 Funktionsumfang von LOSA 14
3.1.1 Aufzeichnungsumfang 15
3.1.2 Regulierung der Aufzeichnungsgranularität 16
3.1.3 Sicherheitskonzept 17
3.1.3.1 Capabilities 17
3.1.3.2 Verschiedene Betriebsmodi 18
3.1.4 Konfigurierungsmöglichkeiten 18
3.2 Bestandteile von LOSA 22
3.2.1 Kernel-Erweiterung 22
3.2.2 Audit-Bibliothek 23
3.2.3 Audit-Ereignis-Konfigurations-Optimierer 23
3.2.3.1 Startoptionen 24
3.2.4 Audit-Dämon 24
3.2.4.1 Standardausgabe über den syslog-Dämon 25
3.2.4.2 Start-Optionen 26
3.2.4.3 Trail-Verwaltung 26
3.2.4.4 Fehlerbehandlung 27
3.2.5 Audit-Konfigurationsinterface 27
3.2.5.1 Aufrufparameter 27
3.2.5.2 Shell-Kommandos 28
3.2.6 Trail-Viewer 29

Kapitel 4 **Implementierung von LOSA** **31**
4.1 Erweiterungen des Betriebssystem-Kernels 31
4.1.1 Der Kernel-Zugangsfunktion `sys_auditcall(...)` 32
4.1.2 Erweiterungen der Prozess-Struktur 33
4.1.2.1 Audit-Struktur 34
4.1.2.2 Preselektionsmaske 35
4.1.3 Audit-ACL 35
4.1.3.1 Die Audit-ACL-Datenstruktur 36
4.1.3.2 Einfügen von Audit-ACEs 37

4.1.3.3 Durchsuchen der Audit-ACL 38
4.1.4 Capabilities 38
4.1.5 Verwaltung der Audit-Daten 39
4.1.5.1 Aufbau des Audit-Puffer 39
4.1.5.2 Generierung eines Audit-Ereignissen 42
4.1.5.3 Speicherung der Audit-Daten 44
4.2 Aufbau des Audit-Trails 45
4.2.2 Ausnahmen bei connect- bzw. accept-Ereignissen 46
4.3 Kommunikation zwischen Audit-Dämon und Steuerungsprogramm 46
4.4 Betriebsmodus 48
4.5 Start-Skript `audit` 49
4.5.1 Init `audit start` 50
4.5.2 Applikation-Wrapper 52
4.6 Ressourcenverbrauch 52
4.6.1 Vergleich der allgemeinen Systemleistung 52
4.6.2 Menge der anfallenden Audit-Daten 53

Kapitel 5 **Zusammenfassung 54**
5.1 Vergleich der vorgestellten Konzepte 54
5.1.1 Aufzeichnungsumfang 54
5.1.2 Konfigurierungsmöglichkeiten 55
5.2 Was bleibt offen? 56

Anhang A **Systemintegration 58**
A.1 Hardware-Voraussetzungen 58
A.2 Software- Voraussetzungen 58
A.3 Installation 58
A.3.1 Install-Skript 59
A.3.2 Manuelle Installation 60

Anhang B **Überwachte Ereignisse 64**
B.1 Systemrufe 64
B.2 Zusätzliche Audit-Events 72
B.3 Applikation-Events 73
B.3.1 login-Event 73
B.3.2 su-Event 74

Anhang C **Technische Beschreibung 75**
C.1 Datenstrukturen 75
C.2 Bibliotheksfunktionen 79
C.3 Grammatiken der Shell- und Konfigurierungskommandos 85
C.4 Dateien von LOSA 87
C.4.1 Konfigurationsdateien 88
C.4.2 Audit-Bibliothek 88
C.4.3 Man-Pages 88
C.4.4 Patches 88
C.4.5 Script-Dateien 89
C.4.6 Utilities 89
C.5 Geänderte Kernel-Quellen 90

Anhang D **Beispielkonfiguration 91**
D.1 Globale Konfiguration: `audit.conf` 91
D.2 Konfiguration des Audit-Dämon: `auditd.conf` 92
D.3 Vergabe von Audit-IDs: `audit_aid.conf` 93
D.4 Einteilung der Audit-Events in Gruppen: `audit_evt.conf` 93
D.5 Konfiguration der Audit-ACL: `audit_acl.conf` 95

Anhang E **Literaturverzeichnis 97**

Abbildungsverzeichnis

Abbildung 1 : BSM-Bestandteile und deren Abhängigkeiten 5
Abbildung 2 : Übersicht Linux-Betriebssystem-Audit 14
Abbildung 3 : Zweistufige Aufzeichnungsgranulierung 17
Abbildung 4 : Detailübersicht des Audit-Dämon 24
Abbildung 5 : Trail-Viewer Hauptfenster 29
Abbildung 6 : Button-Leiste 30
Abbildung 7 : Das Suchdialogfenster 30
Abbildung 8 : Erweiterung des Betriebssystem-Kernel 32
Abbildung 9 : `sys_auditcall(...)` Kommandohierarchie 33
Abbildung 10 : Audit-ACL-Struktur 36
Abbildung 11 : Audit-Puffer 40
Abbildung 12 : Status eines Audit-Record 41
Abbildung 13: Generierung eines Audit-Records 43
Abbildung 14 : Aufbau des Audit-Trails 45
Abbildung 15 : Kommunikation zwischen Audit-Dämon und Steuerungsprogramm 47

Tabellenverzeichnis

Tabelle 1 : Vergleich der generierten Informationen 11
Tabelle 2 : Vergleich der Filter- und Konfigurationsmöglichkeiten 12
Tabelle 3 : Inhalt eines Audit-Records 15
Tabelle 4 : Menge der Audit-Daten bei den Testläufen 53
Tabelle 5 : Vergleich des Aufzeichnungsumfangs 54
Tabelle 6 : Vergleich der Konfigurationsmöglichkeiten 55

Kapitel 1

Einleitung

Es war Anfang der 90ziger Jahre als **Linus Torwalds** die erste Version des Linux-Kernel[1] im Internet der Öffentlichkeit präsentierte. Ziel seiner Arbeit war ein UNIX-ähnlichen Betriebssystem für Intel-basierte PCs, das jedem PC-Besitzer zu einem vernünftigen Preis zur Verfügung steht. Bis dahin war die Verwendung von **UNIXÔ** den Besitzern und Nutzern von teueren Hochleistungsrechnern vorbehalten. Wie bei den kommerziellen UNIX-Systemen, handelt es sich bei Linux um ein echtes multi-tasking, multi-user und multi-session fähiges Betriebssystem. Es erlaubt die gleichzeitige Ausführung von mehreren Programmen, unterstützt die Trennung einzelner Nutzer und gestattet das gleichzeitige Arbeiten mehrerer Nutzer an einem Rechner.

Bereits kurz nachdem die erste Version des Kernel im Internet auftauchte, bildete sich ein große Gemeinde von freiwilligen Programmieren, welche die Entwicklung von Linux vorantrieben. Mittlerweile existieren Portierungen für Digital Alpha™, SunSPARC und Power Macintosh©. Somit ist Linux neben Solaris, AIX und Digital UNIX eines der wenigen Betriebssysteme das sowohl 32-bit als auch 64-bit Architekturen unterstützt. Während Linux bis vor kurzem noch als Hacker-Spielzeug galt, wird es heute in vielen großen und kleinen Unternehmen als eine preiswerte Alternative zu Windows NT oder kommerziellen UNIX-Systemen eingesetzt. Die Leistungsfähigkeit von Linux braucht den Vergleich mit diesen wesentlich älteren und teureren Systemen nicht mehr zu scheuen. Aber das wichtigste Argument für Linux ist wohl die freie Verfügbarkeit des Quellcodes. Dadurch wird vielen Unternehmen die Chance gegeben, das Betriebssystem an die eigenen Anforderungen anzupassen. Gerade auf dem Gebiet der Sicherheit bietet die Offenlegung des Programmtextes die Möglichkeit, das Betriebssystem hinsichtlich möglicher Schwachstellen zu untersuchen. Es bleibt eine Sache der eigenen Einstellung, ob man einem Programm vertraut, dass von einem Unbekannten geschrieben und von vielen unabhängigen Augen geprüft wurde oder einer von einem Softwareunternehmen erstellten Applikation, wo einem versichert wird, dass das von ihnen erstellte Programm geprüft und sicher sei.

Wenn Open-Source eine entscheidende Rolle bei der Verbesserung der Sicherheit von Linux spielt, so gilt dies sicher auch für die Stabilität. Gerade die Stabilität von Linux hat dazu geführt, dass es heutzutage im Serverbereich eine weite Verbreitung findet und immer öfter gegenüber kommerziellen Produkten bevorzugt wird. Neben dem Einsatz im Server-Bereich, etabliert sich Linux immer mehr auf dem Markt der Firewall-Systeme zur Absicherung lokaler Netze. Waren es 1994 noch weniger als 2 Million Linux-Nutzer, so hat sich die Zahl der Nutzer bis 1997 auf 6 Millionen erhöht. 1997 entsprach dies einem Marktanteil von 6,8%. 1998 wuchs die Zahl noch einmal auf 12 Millionen Nutzer. Im selben Jahr hielt Linux im Serverbereich einen Marktanteil von 15,8%, dies entspricht einer Wachstumsrate von 190 Prozent [2]. Heute ist Linux das zweithäufigst eingesetzte Server-Betriebssystem auf dem Weltmarkt, einzig Windows NT wird noch häufiger eingesetzt.

Mit dem zunehmenden Einsatz von Linux im Server- und im Firewall-Bereich wachsen die Anforderungen, die hinsichtlich der Sicherheit an das Betriebssystem gestellt werden. Die Bewertung der Sicherheit von Computer-Systemen erfolgt häufig anhand der vom DoD Computer Security Center im *„Security Requirements for Automatic Data Processing (ADP) Systems“* veröffentlichten Kriterien. Die in diesem Bericht vorgestellten Kriterien zur Bewertung von ADP-Systemen werden in vier Stufen eingeteilt: D, C, B und A; hierarchischen geordnet mit der höchsten Stufe A. In die Stufe D

[1] Wenn hier von Linux gesprochen wird, ist damit der Kernel, das Herz eines jeden UNIX-Betriebssystems gemeint.

werden alle Systeme eingeordnet, die die Anforderungen einer höheren Stufe nicht erfüllen. Die Stufe C ist in die Klassen C1 und C2 unterteilt. Systeme, die in die Klasse C1 eingeordnet werden, müssen zwischen Nutzer und Daten unterscheiden können. D.h. ein Nutzer muss die Möglichkeit haben, seine Projekte oder privaten Informationen vor anderen Nutzern zu schützen. Die Klasse C2 verfeinert die Zugriffskontrolle von C1. Zudem wird in der Stufe C2 eine Aufzeichnung der Login-Prozesse, die Überprüfung (engl. Auditing) von sicherheitsrelevanten Ereignissen und die Isolation von Ressourcen gefordert. Eine Protokollierung und damit eine ausreichende Überprüfung von sicherheitsrelevanten Aktionen ist unter Linux nicht möglich. Damit erfüllt Linux nicht die Anforderungen der Klasse C2. Doch gerade diese Stufe wird oft als Grundvoraussetzung für den Einsatz in sicherheitskritischen Bereichen gefordert.

Der Audit-Prozess in einem sicheren System beinhaltet die Aufzeichnung, die Untersuchung und die Nachprüfung einzelner oder aller sicherheitsrelevanten Aktivitäten auf dem System. Die gewonnenen Informationen können zur Erkennung und zur Abschreckung von Penetrationen in ein Computer-System und zur Identifizierung von Missbrauch auf dem System genutzt werden [5]. Hierbei unterscheidet man host- und netzbasierte Audit-Daten. Die netzbasierten Audit-Daten werden aus den zwischen Rechnersystemen übertragenen Informationen herausgearbeitet. Es werden nicht die Aktionen überwacht, die die Kommunikation initiiert haben oder die durch die übertragenen Informationen ausgelöst wurden. Hierzu werden hostbasierte Audit-Daten benötigt. Sie beinhalten alle Aktionen, die auf einem System ausgeführt wurden, dabei wird nicht unterschieden, ob die Aktion über das Netzwerk oder lokal initiiert wurden. Hostbasierte Audit-Daten werden in der Regel durch ein sogenanntes Betriebssystem-Audit generiert. Das Betriebssystem-Audit ist eine zusätzliche Funktionalität des Betriebssystems und ist meist nicht integraler Bestandteil des Systems.

Für eine spätere Analyse oder eine ausreichende Audit-basierte Überwachung müssen die gesammelten Audit-Daten ein Mindestmaß an Informationen enthalten. Die notwendigen Informationen lassen sich in dem 5-Tupel:

WER, WANN, WIE, WO, RESSOURCE

zusammenfassen.

WER Damit die aufgezeichnete Aktivität einem Initiator zugeordnet werden kann, muss dieser in Verbindung mit dem aufgezeichneten Ereignis im Audit-Record vermerkt werden.

WANN Für die Erkennung von IT-Sicherheitsverletzungen spielt der zeitliche Kontext, in dem die Aktion ausgeführt wurde, ein wichtige Rolle. Eine Vielzahl von Aktivitäten setzt sich aus einer Reihe von sicherheitsunkritischen Einzelaktionen zusammen, die, wenn sie hintereinander ausgeführt werden, eine eindeutige IT-Sicherheitsverletzung darstellen.

WIE Für die Beschreibung des Ereignisses sind eine Reihe von Informationen notwendig, die die Art und Weise der möglichen IT-Sicherheitsverletzung beschreiben.

WO Da die Auswertung der Audit-Daten nicht immer lokal bzw. in einem größeren Netzwerk oftmals zusammenfassend für mehrere Rechner durchgeführt wird, muss im Audit-Record der Name oder ein adäquater Identifier des Rechner, auf dem die Aktion ausgeführt wurde, eingefügt werden. Dieser Rechner wird im folgenden als Wirtsrechner bezeichnet.

RESSOURCE Bei den aufzuzeichnenden sicherheitsrelevanten Aktionen unterscheidet man zwei Arten, Aktionen, die sich auf eine Ressource beziehen (bspw. mittels `open(...)`), und Aktionen, die den aktuellen Systemzustand verändern (bspw. mittels `setuid(...)`). Um später eventuelle Schäden identifizieren zu können, ist es notwendig, dass die betroffene Ressource ausreichend im Audit-Record beschrieben ist.

In dieser Arbeit wird das im Rahmen des Projektes LOSA (**Linux Operating System Audit**) entstandene Audit-Konzept für Linux vorstellt. Das Projekt beinhaltet ein Software-Paket, welches ein Standard-Linux-System um eine Audit-Funktionalität erweitern, wie sie einleitend gefordert wurde.

Als Vorlage für das hier vorgestellte Konzept, diente das BSM (*Basic Security Modul*) von Sun Microsystems für Solaris und das Sicherheitsmodell von Microsoft für Windows NT. Beide Konzepte werden im folgenden Abschnitt genauer vorgestellt. Außerdem beschreibt das Kapitel 2 zwei Konzepte für Linux, die den aktuellen Stand der Protokollierungsmöglichkeiten unter Linux darstellen. Im dritten Abschnitt erfolgt die detaillierte Vorstellung von LOSA. Dazu erfolgt zunächst eine Beschreibung der Leistungsmerkmale und anschließend eine Vorstellung der einzelnen Bestandteile. Im Kapitel 4 wird auf die Implementierung einzelner Komponenten eingegangen. Die Beschreibung dient sowohl als Anwendungs- als auch als Entwicklerdokumentation. Die im Kapitel 4 und im Anhang beschriebenen Einzelheiten sollten einen ausreichenden Überblick geben, um damit arbeiten und um es weiterentwickeln zu können. Im Kapitel 5 erfolgt eine Gegenüberstellung von LOSA mit den in Kapitel 2 vorgestellten Konzepten.

Kapitel 2

Bestandsaufnahme von bestehenden Betriebssystem-Audit-Konzepten

Obwohl Linux heutzutage in vielen sicherheitskritischen Bereichen eingesetzt wird, gibt es fast keine Möglichkeit, Aktionen, die auf einem Linux-Rechner ausgeführt werden, für eine spätere Analyse zu protokollieren. Hierzu wäre ein in den Betriebssystemkern integriertes Audit notwendig. Die vorhandenen Logging-Mechnismen sind zum Großteil applikativ und bei weitem nicht informativ genug [8], [9]. Dieser Abschnitt gibt einen Überblick über die Audit-Module von Solaris und Windows NT und zeigt, warum die für Linux vorhandenen Mechanismen als zu gering eingeschätzt werden müssen.

2.1 Audit-Konzepte in marktgängigen Betriebssystemen

An dieser Stelle erfolgt eine Beschreibung der Audit-Konzepte von SUN Microsystems für Solaris und von Microsoft für Windows NT. Beide Konzepte dienten als Vorlage für LOSA. Die Erläuterung dient gleichzeitig als Einführung für das in Kapitel 3vorgestellte Linux-Betriebssystem-Audit-Konzept.

2.1.1 Das *Basic Security Modul (BSM)* von Solaris

Neben den grundlegenden Sicherheitsmechanismen eines UNIX-Systems stellt Solaris (2.x) mit Hilfe des BSM eine Betriebssystem-Audit zur Verfügung. Das bereitgestellte Audit-System ist eine sicherheitstechnische Erweiterung und nicht integraler Bestandteil von Solaris, d.h. das Betriebssystem ist nicht vom Audit abhängig und kann ohne dieses betrieben werden.

2.1.1.1 Bestandteile des BSM von Solaris

Das BSM setzt sich aus mehreren, aufeinander aufbauenden Bestandteilen zusammen. Die grundlegende Basis bildet die Audit-ID und die Session-ID mit den Funktionalitäten des Kernel-Moduls und des Audit-Dämon. Dabei stützen sie sich zum Teil auf die Informationen, die von den grundlegenden Sicherheitsmechanismen bereitgestellt werden.

Audit-ID

Alle grundlegenden Sicherheitsmechanismen eines UNIX-Systems bauen auf die Nutzer-ID auf. Sie ist zentraler Bestandteil bei jeder Zugangskontrolle und bei der Vergabe von Privilegien. Mittels verschiedener Programme (bspw. `su`) ist es möglich, temporär die Nutzer-ID zu wechseln. Für ein Audit-System ist es wichtig, die Aktionen einem Nutzer eindeutig zuordnen zu können. Aus diesem Grund wird neben der Nutzer-ID eine Audit-ID mitgeführt. Die Audit-ID (`aid`) ist eine nicht veränderbare Nutzeridentifikation. Sie wird beim initialen Anmelden des Nutzers gesetzt und bleibt bis zum Abmelden unverändert. Jeder Prozess, der während einer Nutzer-Session gestartet wird, erbt die Audit-ID von seinem Vaterprozess. Damit kann für jede protokollierte Aktion (Audit-Event) der Nutzer, welcher dieses Aktion initiiert hat, eindeutig identifiziert werden. Bei Solaris ist die Audit-ID gleich der initialen Nutzer-ID.

Session-ID

Neben der eindeutigen Identifikation des Nutzer ist für eine spätere Auswertung der Audit-Daten eine eindeutige Identifikation der Nutzer-Session von fundamentaler Bedeutung. Eine Session beginnt mit dem Anmelden des Nutzers. Zu diesem Zeitpunkt wird neben der bereits erwähnten Audit-ID die Session-ID (`sid`) vergeben. Bei Solaris ist die `sid` gleich der Prozess-ID des Login-Prozesses. Meldet sich der Nutzer ein zweites Mal an das System an, öffnet er eine neue Session und bekommt eine neue Session-ID. Mittels Audit-ID und Session-ID kann bei einer späteren Analyse ein Audit-Event einem Nutzer und einer bestimmten Session zugeordnet werden. Dies ist besonders dann von Bedeutung, wenn nach einer bestimmten Sequenz von Nutzer-Aktionen gesucht wird.

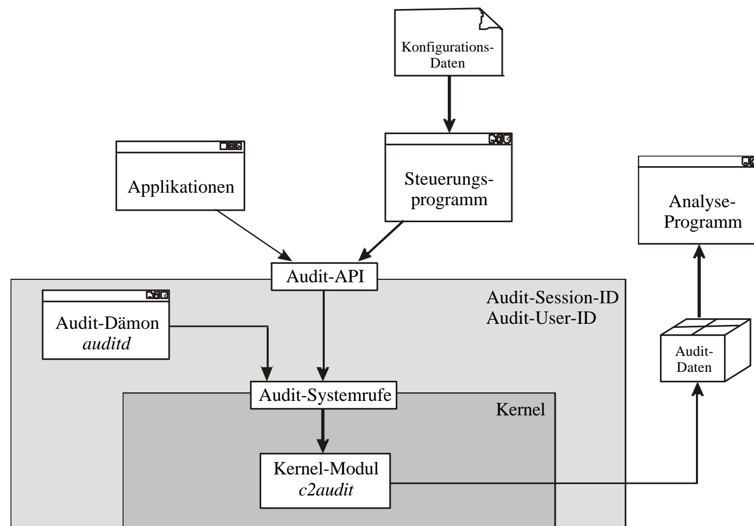

Abbildung 1 : BSM-Bestandteile und deren Abhängigkeiten

Kernel-Modul

Das Kernel-Modul muss beim Starten des Systems eingebunden werden und ist für die Generierung der Audit-Daten verantwortlich. Standardmäßig befinden sich im Kernel zunächst nur sogenannte Stubs, die den Zugang zum Audit-System darstellten. Erst wenn das Kernel-Modul `c2audit` geladen wurde, existieren hinter diesen Stubs Funktionen, die die Audit-Daten generieren. Zusätzlich stellt das Audit-Modul die Funktionen für die Audit-Systemrufe zur Verfügung.

Audit-Dämon

Der Audit-Dämon arbeitet als Nutzerprozess und somit außerhalb des Betriebssystemkerns. Er initiiert die Speicherung der im Kern generierten Audit-Records und die Verwaltung der Audit-Dateien. Anhand von Limits entscheidet der Audit-Dämon, wann eine neue Datei geöffnet werden muss. Die Verwendung eines zusätzlichen Nutzerprozesses hat den Vorteil, dass die Fehlerbehandlung nicht im Kern durchgeführt werden muss, was sich als äußerst schwierig erweisen würde. Ferner obliegt damit die Kontrolle des Audit-System einem Nutzerprozess und nicht dem Betriebssystemkern.

2.1.1.2 Funktionsumfang des BSM

Im Gegensatz zum Audit-System von Windows NT stellt das BSM von Solaris eine reine Kernel-Erweiterung dar. Die Generierung der Audit-Daten erfolgt im Kernel. Das Erzeugen eines Audit-Records wird mit dem Aufruf eines Systemrufen initiiert. Durch die Protokollierung der Systemrufe können sowohl alle Änderungen am System (bspw. `setuid(...)`, `stime(...)`, ...) als auch alle Änderungen am Dateisystem (bspw. `open(...)`, `unlink(...)`, ...) protokolliert werden.

Wie bei jedem UNIX-System besteht auch bei Solaris die Möglichkeit, Applikationen zu schreiben, die sicherheitsrelevante Aktionen ausführen, aber nicht vom BSM-Modul erfasst werden (bspw. `ssh` (*Secure Shell*), `xdm` (*grafischer Login*), ...). Aus diesem Grund wird durch das Audit-API eine

allgemeine Schnittstelle angeboten, über die jede Applikation Audit-Events generieren und somit in den allgemeinen Audit-Prozess einbezogen werden kann.

Aufbau der Audit-Records

Jedes protokollierte Ereignis wird in einem Audit-Record abgelegt. Die einzelnen Records werden wiederum in sogenannte Token unterteilt. Für jeden Record existiert ein Header-Token und eine freie Anzahl von zusätzlichen Token. Sie dienen der Beschreibung des aufgezeichneten Ereignisses. Neben festvorgeschriebenen Token besteht die Möglichkeit, je nach Konfiguration, zusätzliche Token durch das Audit-System einfügen zu lassen. Diese enthalten weitergehende Informationen, die bei einer späteren Analyse von Nutzen sein können. Die Unterteilung der Records in einzelne Token ermöglicht eine effiziente Generierung der Audit-Daten und eine platzsparende Speicherung auf einem permanenten Medium.

Im Header-Token werden folgende Informationen hinterlegt:

- Zeitstempel,
- Event-ID und
- Event-Modifier.

Für die Beschreibung des Initiators werden im Subject-Token folgende Informationen hinterlegt:

- Audit-ID,
- relative/effektive Nutzer/Gruppen-ID,
- Prozess-ID,
- Session-ID,
- Terminal-ID und
- Herkunftsrechner.

Bezieht sich das protokollierte Ereignis auf eine Ressource werden in einem Path- und Attribut-Token folgende Daten abgelegt:

- Dateisystem-/ Device-ID,
- Inode,
- Pfadname,
- Nutzer-ID und Gruppen-ID des Besitzers,
- Zugriffsrechte und
- Ressourcentyp.

Die Informationen für den Status der Aktivität werden in einem Return- bzw. Exit-Token hinterlegt. Zusätzlich kann für Ereignisse (`exec()` bzw. `execve()`), bei denen die Argument oder die Prozessumgebung von Interesse ist, ein Argument- und ein Environment-Token angefordert werden. Darüber hinaus kann für Testzwecke mittels des Sequenz-Token jeder Record mit einer eindeutigen, fortlaufenden Nummer versehen werden.

Eine genaue Beschreibung der einzelnen Token und deren Inhalt findet man in [7] in der Datei `/net_os/src/uts/common/c2/audit_token.c`.

Filter-Möglichkeiten

Um die Menge der generierten Audit-Daten in Grenzen zu halten, ist ein selektive Aufzeichnung wünschenswert. Das BSM von Solaris bietet drei Möglichkeit die Aufzeichnungsgranularität einzustellen. Innerhalb des Audit-Systems erfolgt die Selektierung über:

- systemweite und nutzerbezogene Audit-Flags und Präfixe,
- Policy-Flags und

- Prozesspräselektionsmasken.

Audit-Flags

Mittels der Audit-Flags kann die Aufzeichnung von Audit-Events aktiviert bzw. deaktiviert werden. Die Audit-Flags entsprechen sogenannten Audit-Klassen. Um die Administration zu vereinfachen werden unter Solaris die Audit-Events in Audit-Klassen eingeteilt. Die Verwaltung der Audit-Klassen erfolgt in ein 32-Bit-Maske. Damit können 32 + 2 verschiedene Audit-Klassen unterschieden werden[2].

Die Audit-Flags können sowohl global als auch Nutzer-bezogen gesetzt werden. Damit kann die Aufzeichnung von Audit-Klassen gezielt aktiviert oder deaktiviert werden. Außerdem kann mittels Präfixe entsprechend dem Ergebniswert gefiltert werden. Unterscheidungen sind für erfolgreiche (+), fehlgeschlagene (-) sowie beide Fälle (kein Präfix, none) vorgesehen.

Policy-Flags

Die Audit-Policies werden zur Regulierung des Umfangs der zu generierenden Audit-Daten verwendet. Hier können die zuvor erwähnten Record-Token ausgewählt werden. Die Auswahl gilt systemweit für alle Audit-Klassen und für alle Nutzer.

Prozesspräselektionsmasken

Für jeden Nutzer, der sich an das System anmeldet, wird eine Prozesspräselektionsmaske gebildet. Diese setzt sich aus den systemweiten und nutzerbezogenen Audit-Flags zusammen und wird in die Task-Struktur des Login-Prozesses angehängt. Jeder neue Prozess erbt die Maske von seinem Vater-Prozess, sie gilt damit bis zum Abmelden des Nutzers. Die Prozesspräselektionsmaske kann über die Konfigurationsdateien oder zur Laufzeit mittels des Programms `auditcontrol` modifiziert werden.

2.1.2 Ereignis-Protokollierung unter Windows NT

Windows NT 3.51 (und 4.0) Workstation und Server in Verbindung mit dem Service Pack 3 und unter Nutzung des NT-Dateisystems NTFS erfüllen die Kriterien des C2-Standards, welcher unter anderem ein Auditing verlangt.[6]

Im Gegensatz zur Vergabe von Benutzerkonten sowie der Zuteilung von Benutzerrechten und Objektrechten, wie sie in einem UNIX-System vergeben werden können, wird unter Windows NT die Zugriffskontrolle über die sogenannte ACL (access control list) gesteuert. Ein Anwender darf eine beabsichtigte Aktion auf einem Objekt nur durchführen, wenn dessen ACL ihm (direkt) oder einer seiner Gruppen (indirekt) den entsprechenden Zugriff gestattet. Neben der reinen Zugriffskontrolle wird die ACL auch zur Einstellung des Audit-Systems verwendet.[6]

2.1.2.1 Bestandteile des Windows NT Audit

Da der Aufbau des Windows NT Audit-Systems für das LOSA-Projekt nicht von Interesse ist, erfolgt an dieser Stelle nur eine kurze Vorstellung der wichtigsten Komponenten.

Eindeutige Identifizierung

Windows NT in der Versionen 3.51 und 4.0 Workstation oder Server ist keine echtes Multi-Session-Betriebssystem[3]. Es ist nicht möglich, dass zwei Nutzer zur gleichen Zeit an einer Maschine arbeiten. Alle Prozesse, die auf einer Windows NT Maschine laufen, gehören entweder zum Betriebssystem selbst oder dem gerade angemeldeten Nutzer. Aus diesem Grund kann bei Windows NT auf eine Audit-ID und eine Session-ID verzichtet werden. Die eindeutige Identifizierung der protokollierten Ereignisse erfolgt über die Nutzer-ID und die Prozess-ID.

[2] Zwei Metaklassen kein Bit bzw. alle 32 Bits gesetzt.

[3] Eine Ausnahme bildet der Windows NT 4.0 Terminalserver, hierbei handelt es sich um eine Multi-User- und Multi-Session-Plattform von Microsoft, die aber hier nicht betrachtet wird.

Ereignisanzeige
Eine Überprüfung der angefallenen Ereignisse kann unter Windows NT mit dem Programm `Ereignisanzeige` erfolgen. Hierbei handelt es sich um ein Programm mit einer grafischen Benutzeroberfläche und diversen Unterdialogen mit deren Hilfe einzelne Ereignisse betrachtet und gesucht werden können.

Die `Ereignisanzeige` stellt neben den reinen sicherheitsrelevanten Ereignissen noch die Rubriken:

- *System*: Ereignisse der Systemkomponenten und
- *Anwendung*: applikative Ereignisse

dar.

Konfiguration
Die Konfiguration des Audit-Systems erfolgt an zwei Stellen. Neben der globalen Aktivierung der Protokollierung von Objektzugriffen im Benutzermanager, müssen die gewünschten Dateien und Verzeichnisse im Dateimanager bzw. Explorer für die Überwachung konfiguriert werden. Die zweite Konfigurationsmöglichkeit stellt ein wesentlicher Unterschied in den Filtermöglichkeiten zwischen Windows NT und UNIX dar, denn mittels des ACL-Mechanismus kann die Protokollierung gezielt auf einzelne Dateien bzw. Verzeichnisse abgestimmt werden. Wobei die Konfiguration etwas umständlich und zeitaufwendig ist, da für jedes zu konfigurierende Objekt eine ganze Anzahl von Dialogen geöffnet werden muss.

2.1.2.2 Funktionsumfang des Windows NT Audit

Wie beim BSM von Solaris werden alle aufgezeichneten Ereignisse mit einem Zeitstempel, der eindeutigen Nutzeridentifikation, dem Wirtsrechner, einer Event-ID und einer Event-Kategorie versehen. Windows NT ist ebenso in der Lage, die erfolgreichen als auch die erfolglosen Ereignisse zu protokollieren.

Neben den für alle Events gültigen Daten werden, wie bei Solaris, Ereignis-spezifische Einträge hinzugefügt. So wird unter anderem jede betroffene Ressource oder bei der Ausführung eines Programms der Programmname und evtl. die neue Prozess-ID hinterlegt.

Globale Überwachungsrichtlinien
Die Konfiguration der globalen Überwachungsrichtlinien erfolgt im Benutzermanager. Hier können folgende Ereignisse aktiviert werden:

- An- und Abmelden
- Datei- und Objektzugriffe
- Verwendung von Benutzerrechten
- Benutzer- und Gruppenverwaltung
- Sicherheitsrichtlinienänderungen
- Neustarten und Herunterfahren des Systems
- Prozessverfolgung

Für jedes Event kann unterschieden werden, ob nur die erfolgreichen, nur die fehlgeschlagenen, generell oder gar nicht protokolliert werden soll.

Datei- und Verzeichnisüberwachung
Neben der globalen Aktivierung der Dateizugriffe kann im Explorer für jede Datei oder jedes Verzeichnis der zu protokollierende Zugriff noch einmal genauer spezifiziert werden. Dabei werden folgende Zugriffsarten unterschieden:

- Lesen,
- Schreiben,
- Ausführen,
- Löschen,
- Änderungen an die Zugriffsrechten oder
- Besitz übernehmen.

Für Verzeichnisse kann zusätzlich angegeben werden, ob sich die angegebenen Werte auch auf die Unterverzeichnisse und die Dateien darin beziehen sollen. Wie bei den globalen Richtlinien kann auch hier zwischen erfolgreichen und erfolglosen Aktionen unterschieden werden.

Sonstige Überwachungen

Neben dem Dateimanager verfügen auch andere Systemprogramme über Dialoge zur Überwachung, dazu zählt:

- der Druckmanager,
- der Registereditor und
- die Ablagemappe.

Zusätzlich zu den vom Betriebssystem bereitgestellten Überwachungsmöglichkeiten können auch Applikationen eigene Ereignisse definieren und im Ereignisprotokoll eintragen.

2.2 Bestehende Konzepte bzw. Ansätze für Linux

Die im Abschnitt 2.1 vorgestellten Audit-Konzepte sind zur Überwachung der auf dem Rechner ausgeführten Aktionen implementiert und sind fest in das Betriebssystem integriert. Ziel dieser Arbeit ist es, ein solches Konzept für Linux zu erstellen. Denn dieser Abschnitt wird zeigen, dass es im Moment für Linux nichts Vergleichbares existiert. Die hier vorgestellten Konzepte bieten zwar ein Mindestmaß an Funktionalität, jedoch erfüllt sie nicht die Anforderungen, die an ein Audit-System gestellt werden.

2.2.1 Der Syslog-Mechanismus

Beim Syslog-Mechanismus handelt es sich um eine zentrale Einrichtung, die von verschiedenen Applikationen zur Ausgabe genutzt werden kann. Bspw. Dämon-Prozesse, welche kein Kontrollterminal besitzen, auf welches sie ihre Meldungen ausgeben können. Ebenso hat der Kernel kein solches Terminal. Alle Nachrichten, die vom Kernel oder von einem Dämon-Prozess erzeugt werden, könnten von ihnen entweder in eine Datei oder auf die Systemkonsole ausgegeben werden. Beide Möglichkeiten stellen für einen Systemadministrator ein eher störendes als nützliches Feature dar.

Aus diesem Grund wurde für BSD-Unix eine allgemeine Struktur entwickelt, die es möglich macht, Nachrichten von verschiedenen Prozessen an einem zentralen Ort zu verwalten. Hierbei handelt es sich um den Syslog-Dämon, der auch für Linux verfügbar ist und von den meisten Distributionen standardmäßig installiert wird.

Das Erzeugen von Nachrichten ist auf drei verschiedene Arten möglich:

1. Funktionen im Kernel können `printk(...)` aufrufen, um Nachrichten zu generieren. Diese Möglichkeit wird gewöhnlich vom Kernel für Fehlermeldungen oder von einem Gerätetreiber für Statusmeldungen genutzt.
2. Ein Benutzerprozess (Dämon) verwendet `syslog(...)`, um eine Nachricht an den Syslog-Dämon zu senden.
3. In einem TCP/IP-Netzwerk kann jeder Benutzerprozess Nachrichten auf den *UDP-Port 514* schicken. Dazu ist jedoch etwas Netzwerkprogrammierung nötig.

Die Konfiguration des `syslog`-Dämon erfolgt mittels einer Konfigurationsdatei, normalerweise ist dies `/etc/syslog.conf`. In dieser Datei kann festgelegt werden, wohin die verschiedenen Nachrichten gesendet werden sollen. So kann eine dringende Nachricht auf der Systemkonsole ausgegeben und/oder eine E-Mail an eine bestimmte Person gesendet werden, während eine Warnung lediglich in eine Log-Datei geschrieben wird.

Mittels des Syslog-Dämon ist jedoch kein Betriebssystem-Audit an sich möglich. Die meisten Nachrichten werden auf der Anwendungsebene generiert und nicht wie für ein Audit notwendig, im Kernel bei der Abarbeitung von Systemrufen erzeugt. Allerdings wird eine zentrale Einrichtung geboten, mit deren Hilfe die von verschiedenen, sicherheitsrelevanten Programmen (bspw. **`su`**, **`login`**, **`cron`**, ...) erzeugten Nachrichten an einem Ort verwaltet werden können.

2.2.2 Der Audit-Dämon vom H.E.R.T.

Bei dem hier erläuterten Audit-Dämon handelt es sich um eine Implementierung von M.Wolf vom H.E.R.T.. Die derzeit aktuellste Implementierung ist für die Kernel-Version 2.0.38, eine Portierung auf die aktuelle Version 2.2.x oder 2.4.x ist noch nicht in Sicht, so dass für den Einsatz des Audit-Dämon auf eine entsprechend alte Version des Kernel zurückgegriffen werden muss[4].

Das Paket setzt sich aus einem Kernel-Patch und dem Audit-Dämon zusammen. Beide Komponenten müssen auf dem System installiert werden. Zusätzlich muss der Kernel nach dem Einspielen des Patches neu übersetzt werden. Anschließend kann mittels des Audit-Dämon `auditd` das Betriebssystem-Audit aktiviert werden. Der Dämon-Prozess kann nur mittels `kill` deaktiviert werden. Eine Re-Konfiguration zur Laufzeit ist nicht möglich.

Überwachte Systemrufe

Das Audit-Konzept bietet die Möglichkeit, folgende Systemrufe zu überwachen:

- `execve()`, dient zum Ausführen von Programmen,
- `open()`, dient zum Öffnen und zum Anlegen von Dateien,
- `modint()`, dient zum Laden von Kernel-Modulen,
- `setuid()`, setzt die Nutzer-ID des aktuellen Prozesses,
- `listen()`, dient zum Einrichten einer Socket-Warteschlage,
- `connect()`, baut eine Netzwerkverbindung zu einem entfernten Rechner auf und
- `accept()`, dient zum Annehmen von Netzwerkverbindungen.

Allerdings werden generell nur die erfolgreichen Aufrufe protokolliert. Es existiert keine Möglichkeit, Systemrufe, die fehlgeschlagen sind, aufzuzeichnen. Da die Aufzeichnung von fehlgeschlagenen Systemrufen einen wesentlich größeren Programmieraufwand bedeutet, die Menge der zu protokollierenden Daten um ein Vielfaches größer ist und das zugrundeliegend Konzept hierfür nicht geeignet scheint, ist mit einer dahingehenden Erweiterung des Konzeptes nicht zu rechnen.

Filtermöglichkeiten

Die Konfiguration des Betriebssystem-Audit erfolgt mittels einer Datei, die, wenn man den Quellen des Dämons nicht verändert, unter dem Namen `audit.conf` im Verzeichnis `/etc/security/` abgelegt werden muss. In dieser Datei werden die Filterregeln festgelegt. Für jeden aufzuzeichnenden Systemruf muss eine Zeile der Form:

```
log:[flag]:[pname]:[pid]:[uid]:[file]
```

angelegt werden.

[4] Dies war der Stand des Projektes Anfang des Jahres 2000. Momentan ist eine Portierung für die Kernel-Version 2.2.x verfügbar. Jedoch hat sich am Funktionsumfang des Projektes nichts geändert.

- `flag`, bezeichnet den aufzuzeichnenden Systemruf,
- `pname`, bezeichnet den Programmnamen in dessen Kontext das Event auftreten soll,
- `pid`, kennzeichnet die Prozess-ID in dessen Kontext das Event auftreten soll,
- `uid`, ist die Nutzer-ID, welcher das Event generiert soll und
- `file`, bezeichnet die Archivdatei, in welcher die Daten abgelegt werden sollen.

Zeitstempel

Erfolgt der Aufruf des Audit-Dämon mit der Option `-t`, wird in der Archivdatei jeder abgelegt Eintrag mit einem Zeitstempel versehen.

2.3 Vergleich der Audit-Funktionen

Obwohl die Audit-Konzepte von Solaris, Windows NT oder Linux sehr verschieden sind, kann man die zur Verfügung gestellten Informationen sehr wohl vergleichen. In der Einleitung wurde erwähnt, welche Informationen für eine wirksame Überwachung zwingend notwendig sind. In diesem Abschnitt erfolgt eine Gegenüberstellung des BSM von Solaris, dem Event-Log-Mechanismus von Windows NT und dem Audit-Dämon vom H.E.R.T anhand der im Kapitel 1geforderten Informationen. Auf den im Abschnitt 2.2.1 erwähnten Syslog-Dämon wurde in der Tabelle verzichtet, da dieser die wenigsten der dort geforderten Punkte erfüllen kann.

2.3.1 Aufzeichnungsmöglichkeiten

	Solaris	Windows NT	Linux Audit vom H.E.R.T
	Wer		
Eigentümer	Audit-ID	User SID	-
Privilegien	Reale/effektive Nutzer/Gruppen-ID	Bei Objektzugriffen	Reale Nutzer-ID
System-Identifikation	Prozess-ID, Session-ID	User SID, Anmelde ID	Prozess-ID
Login-Terminal	Terminal-ID	-	Programmname
Programmname	Durch Verfolgung von `fork()`/`exec()`	Durch detaillierte Verfolgung	-
Herkunftsrechner	IP-Adresse	Domäne, Arbeitsstation	-
	Wann		
Zeitstempel	Systemzeit	Systemzeit	Systemzeit
	Wie		
Ereignistyp	Event-ID	Event-ID	Mittels separater Log-Datei
Ereignissubtyp	Ereignis-Modifier	-	-
Ereignisklasse	Lt. Systemfestlegung	Event-Kategory	-
Ereignisstatus	Status	Event-Typ	(Immer erfolgreich)
Argumente	Parameter	-	-
	Wo		
Wirtsrechner	Mittels `gethostname()`	Computername	Mittels `gethostname()`
	Ressource		
Name	Pfadname	Objekt-Name	Pfadname
Typ	Typ	Objekt-Typ	Mittels `stat()`
Zugriffsrechte	Zugriffsrechte	Mittels extra Systemruf	Mittels `stat()`
Besitzer	Eigentümer/ Benutzergruppe	Mittels extra Systemruf	Mittels `stat()`
System-Identifikation	Dateisystem, Gerätenummer, Inode	Mittels extra Systemruf	Mittels `stat()`
Lokation	Mittels `mount()`	Objekt-Server	

Tabelle 1 : Vergleich der generierten Informationen

Der in Tabelle 1 dargestellte Vergleich bezieht sich auf die Informationen, die aus den Audit-Dateien bzw. Protokolldateien gewonnen werden können. Hierbei handelt es sich nicht nur um den Inhalt eines Records, unter Umständen kann es notwendig sein, eine Vielzahl von Records auszuwerten oder zusätzliche Programme einzusetzen, um die hier darstellten Informationen zu gewinnen.

Neben den in Tabelle 1 dargestellten Informationen interessieren häufig globale Ereignisse, die nicht auf einen Systemrufe zurückzuführen sind oder besser direkt eingetragen werden sollten. Hierzu zählen Ereignisse wie Reboot, System herunterfahren, Login, Logout usw..

Während bei Windows NT diese Ereignisse direkt eingetragen werden, besteht bei Solaris und bei Linux das Problem, dass solche Ereignisse durch Fremdprogramme verursacht werden können. Obwohl das BSM von Solaris diese Ereignisse nicht direkt aufzeichnet, ist es mittels des Audit-API leicht möglich, die Fremdprogramme dahingehend zu verändern, dass sie in den Audit-Prozess mit einbezogen werden. Der Audit-Mechanismus vom H.E.R.T. bietet keine Schnittstelle für externe Applikationen, jedoch ist es mittels des Syslog-Dämon möglich, gerade solche globalen Ereignisse aufzuzeichnen. Das PAM-Moduls zeichnet eine Vielzahl globaler sicherheitsrelevanter Ereignissen mittels des Syslog-Dämon auf. Allerdings hat man hiermit die Zentralität verloren. Man ist nun gezwungen, sowohl die Audit-Dateien als auch die log-Dateien auszuwerten.

2.3.2 Filter- bzw. Konfigurationsmöglichkeiten

Neben dem Aufzeichnungsumfang spielt gerade die Möglichkeit der Selektierung eine entscheidende Rolle, in wieweit ein Audit-System einsetzbar ist. Sicherlich ist es nicht schwer, alle Aktionen auf einem System aufzuzeichnen. Die gewonnen Informationen haben jedoch einen relativen geringen Wert, da die Auswertung schwer oder gar unmöglich wird. Ein Audit-System, dass bereits bei der Generierung entscheidet, wie mit diesem Ereignis verfahren werden soll, ist weitaus flexibler einsetzbar.

Ein Großteil der überflüssigen Informationen wird bereits bei der Implementierung durch das Weglassen der Funktion herausgefiltert. Jedoch existieren immer wieder Situationen, in denen ein Ereignis an der einen Stelle interessant und an der anderen Stelle völlig überflüssig ist. In diesem Abschnitt erfolgt eine Gegenüberstellung der einzelnen Filter und Konfigurationsmöglichkeit.

	Solaris	Windows NT	Linux Audit vom H.E.R.T
Filter			
Nutzer-ID	Audit-ID	Nutzername	Reale Nutzer-ID
Prozess-ID	Prozess-ID	Prozessverfolgung	Prozess-ID
Event-ID	Event-Klassen	-	Systemruf
Ergebnistyp	Erfolgreich, Fehlgeschlagen, beides	Erfolgreich, Fehlgeschlagen, beides	Erfolgreich
Ressource	-	Mittels ACL	-
Programmname	-	-	Pfadname
Kombination	Nutzer-ID / Event-Klasse Prozess-ID / Event-Klasse	Nutzer-ID / Event-ID Ressource / Event-ID Nutzer-ID / Ressource / Event-ID	Nutzer-ID / Event Prozess-ID / Event Programmname / Event (sowie alle weitern Kombinationen)
Konfiguration			
Mittels	Konfigurationsdateien	Dialogfenstern	Konfigurationsdatei
Rekonfiguration zur Zulaufzeit	Mittels `auditcontrol`	Mittels Dialogfenstern	-

Tabelle 2 : Vergleich der Filter- und Konfigurationsmöglichkeiten

Der Filtermechanismus von Windows NT ist hinsichtlich einer späteren Analyse oder Weiterverarbeitung besonders leistungsfähig. Windows NT arbeitet mit der bereits erwähnten ACL. Die Konfigurationsmöglichkeiten sind äußert umfangreich. Allerdings besteht bei einer feingranularen Abstimmung das Problem, dass sich die Gesamtperformanz des Systems umgekehrt proportional zur Größe der ACL verhält. Bei Solaris hat man bewusst den Kompromiss gewählt, auf eine Anzahl von Filtermechanismen zu verzichten, um die Performanz des Systems nicht zu stark zu beeinflussen. Obwohl der Audit-Mechanismus vom H.E.R.T. in dieser Tabelle ein relativ gutes Bild macht, ist dessen Leistungsfähigkeit nicht mit dem BSM von Solaris und dem ACL-Mechanismus von Windows NT vergleichbar. Die Anzahl der überwachten Ereignisse ist so gering und die Implementierung ist so gewählt, dass man hier getrost alles filtern kann.

Eine weitere wichtige Eigenschaft ist die Konfigurationsmöglichkeit. Windows typisch werden alle Einstellung in Dialogen vorgenommen. Während dies für einzelne Ereignisse recht komfortable ist, hat man bei einem größeren System das Problem, dass die Anzahl der zu bearbeitenden Dialoge schnell ausufert. Wie für eine UNIX-System typisch werden bei Solaris und Linux alle Einstellungen in zentralen Konfigurationsdateien vorgenommen. Sowohl das BSM als auch das Audit-System von Windows NT können zur Laufzeit rekonfiguriert werden, was besonders bei Tests oder bei der Einrichtung des Systems wichtig ist. Der Audit-Dämon vom H.E.R.T. kann nicht einmal deaktiviert werden (außer mittels `kill -9 [pid]`).

Kapitel 3

LOSA – Linux Operating System Audit

Beim LOSA-Konzept handelt es sich um eine Implementierung eines Betriebssystem-Audit für Linux, ähnlich dem BSM von Solaris. Ziel des Projektes war es, ein Audit-System zu entwickeln und zu implementieren, das in der Lage ist, alle sicherheitsrelevanten Aktionen auf einem Linux-System zu protokollieren, so dass für eine spätere Analyse oder Audit-basierte Überwachung ein Mindestmaß an Informationen zur Verfügung steht. Das Funktionskonzept orientiert sich stark am BSM von Solaris. Bei beiden Systemen handelt es sich um UNIX-Systeme, so dass die Funktionsweise und der Aufbau des Kernels vergleichbar sind.

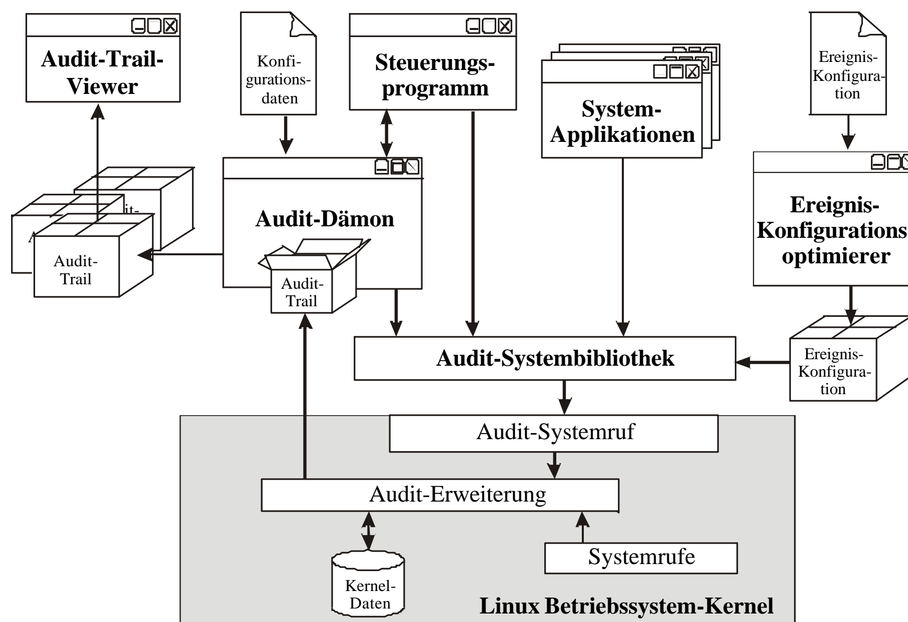

Abbildung 2 : Übersicht Linux-Betriebssystem-Audit

Abbildung 2 zeigt die einzelnen Komponenten von LOSA. Den Kern des Audit-Systems bilden die Kernel-Erweiterung, der Audit-Dämon und das Steuerungsprogramm. Der Trail-Viewer dient der einfachen Analyse der Audit-Trails. Neben diesen Komponenten existieren noch einzelne Erweiterungen in sicherheitsrelevanten Applikationen, wie `login`, `kdm` oder `su` und der Ereignis-Konfigurations-Optimierer, welcher zur Durchsetzung eines vereinfachten ACL-Mechanismus notwendig ist.

3.1 Funktionsumfang von LOSA

Wie bereits einleitend erwähnt, war das Ziel von LOSA ein ausreichende Protokollierung aller sicherheitskritischen Aktionen auf einem Linux-Rechner. Dieser Abschnitt zeigt die Fähigkeiten von LOSA. Obwohl das Konzept stark an das BSM von Solaris angelehnt ist, ist es mit ihm nicht gleichzusetzen. Bei einer eingehender Beschäftigung mit dem Audit-System von Solaris zeigten sich Defizite, die beim Linux-Audit vermieden oder beseitigt werden sollten. Ein Vergleich von LOSA mit den im Abschnitt 2.1 vorgestellten Konzepten von Solaris und Windows NT erfolgt im Kapitel 5.

3.1.1 Aufzeichnungsumfang

Der Aufzeichnungsumfang gibt Auskunft über die Vielfalt der aufgezeichneten Audit-Daten. Bei einem Betriebssystem-Audit werden im allgemeinen Daten aus dem Betriebssystem-Kern aufgezeichnet. Dabei handelt es sich um die Argumente und die Informationen, die in einem Systemruf anfallen. Zusätzlich können Applikationen Audit-Daten generieren, jedoch ist dies nicht zwingend erforderlich für ein Betriebssystem-Audit. Ein Beispiel wäre das Login-Event: Wenn sich ein Nutzer an das System anmeldet, wird durch das login-Programm ein spezielles Event generiert. Jedoch könnte man das Anmelden des Nutzers auch anhand folgender Signatur erkennen:

```
setaid(uid) ⇨ setuid(uid) ⇨ chdir(/home/user) ⇨ execve(bash)
```

Allerdings wäre dies äußert umständlich und die Erkennung von fehlgeschlagenen Versuchen wäre so gut wie unmöglich. Aus diesem Grund erzeugt das login-Programm ein Login-Event.

Neben dem gerade erwähnten Login-Event werden die folgenden Kernel-unabhängigen Ereignisse aufgezeichnet:

- Systemstart, Zeitpunkt des Systemstarts
- su, temporärer Wechsel der Nutzerrechte

Linux selbst verfügt über 190 Systemrufe (einige sind nicht implementiert), davon sind jedoch viele für ein Audit uninteressant. LOSA überwacht 52 der 190 Systemrufe, dabei handelt es sich um alle systemverändernden, alle systembeeinflussenden Systemrufe sowie um alle Systemrufe die Dateisystemressourcen verändern. Eine genaue Übersicht aller überwachten Systemrufe befindet sich im Anhang B.

	Beschreibung
`date`	einen Zeitstempel
`aid`	Eindeutige Nutzer-ID
`ruid`	reale Nutzer-ID
`rgid`	reale Gruppen-ID
`euid`	effektive Nutzer-ID
`egid`	effektive Gruppen-ID
`hostname`	Name und die Domäne des Wirtsrechners
`tty`	Kontrollterminal, von dem die Aktion initiiert wurde
`session`	Session in dessen Kontext die Aktion initiiert wurde
`from`	IP-Adresse des Rechners von dem sich Initiator angemeldet hat
`pid`	Prozess-ID in dessen Kontext die Aktion ausgeführt wurde
`status`	Finaler Ausführungsstatus
`event`	ID des aufgezeichneten Ereignisse
`arg1`	Platzhalter für ein Argument, das dem Systemruf übergeben wurde
`arg2`	Zweiter Platzhalter für ein Argument
`env`	Environment, welches dem Systemruf `execve(...)` übergeben wurde
`file`	Pfadname der Ressource auf den sich der Systemruf bezogen hat an
`owner`	Nutzer-ID des `file`-Besitzers
`gowner`	Gruppen-ID des `file`-Besitzers
`dev`	Gerätenummer auf dem sich `file` befindet
`perm`	Zugriffrechte und Typ von `file`
`inode`	Eindeutige Identifikation von `file` auf dem Gerät
`file2`	Dateiname einer zweiten Ressource, wenn der Systemruf sich auf zwei Ressourcen bezieht

Tabelle 3 : Inhalt eines Audit-Records

Jedes aufgezeichnete Audit-Event wird in einem sogenannten Audit-Record abgelegt. Im Gegensatz zu Solaris wird für alle Events das gleiche Record-Format verwendet. Es ist nicht möglich, den Aufbau des Audit-Records zu beeinflussen. Ein Record kann folgende Informationen enthalten:

Die ersten 13 Einträge (bis `event`) werden bei jedem Record gefüllt, sie entsprechen in etwa einem festem Header, der das Ereignis und dessen Kontext spezifiziert.

Bei keinem Ereignis wird der Record komplett gefüllt, da einige Ereignisse spezifische Informationen enthalten, die für eine spätere Analyse interessant sind, aber von keinem andere Systemruf generiert werden. Ein Beispiel hierfür wäre `execve(...)`, für diesen Systemruf existiert der Eintrag `env`. Bei keinem anderen Systemruf existieren ähnlich Informationen. Desweiteren werden die Daten zum Eintrag `file` nur gefüllt, wenn der Systemruf eine Ressource des Dateisystems manipuliert. Eine genaue Übersicht, in wie weit die variablen Einträge gefüllt werden, ist im Anhang B aufgelistet.

3.1.2 Regulierung der Aufzeichnungsgranularität

Neben der Möglichkeit, sicherheitsrelevante Aktionen aufzuzeichnen, benötigt man ebenfalls die Möglichkeit, die Aufzeichnungsgranularität zu steuern. Denn obwohl von LOSA nur ein Teil der Systemrufe überwacht wird, kann die Menge der Audit-Daten schnell wachsen. Um dem entgegenzuwirken benötigt man einen Mechanismus, der bereits beim Erzeugen der Audit-Daten entscheidet, ob das Ereignis aufgezeichnet werden soll oder nicht.

Gerade im Multi-User-Betrieb existieren eine Vielzahl von verschiedenen Nutzerumgebungen. Ein Systemadministrator kann nicht für jeden Nutzer einzeln die Systemrichtlinien festlegen. Mitunter existieren Verzeichnis oder Daten, die bei jedem Nutzer vorhanden, jedoch für ein Audit-basierte Analyse uninteressant sind. Bspw. ist `Netscape`, eine Anwendung, die von fast jedem Nutzer mindestens einmal am Tag verwendet wird. `Netscape` legt für jeden Nutzer einen eigenen Cache für Internet-Seiten an, um den Zugriff zu beschleunigen. Während einer solchen Netscape-Session wird unzählige Male im Verzeichnis `/home/nutzer/.netscape/cache` schreibend bzw. lesend auf verschiedene Dateien zugegriffen. Für eine spätere Analyse sind diese Ereignisse völlig irrelevant und blähen den Audit-Trail nur unnötig auf.

Notwendig sind Mechanismen, welche die Generierung der Audit-Events in Abhängigkeit von verschiedenen Kontextbedingungen ermöglichen. Ein UNIX-System ist Prozess-basiert, d.h. es können nur Aktionen ausgeführt werden, die im Kontext eines Prozess bzw. eines Nutzers stehen. Ausgehend von dieser Tatsache werden die Informationen, die zur Durchsetzung der Selektion der aufzuzeichnenden Ereignisse notwendig sind, in die Datenstruktur des Prozesses eingetragen. Damit kann theoretisch für jeden Prozess eine andere Selektionsmaske angeben werden.

Da aber die Prozess-ID nicht vorhersagbar und damit keine statische Konfiguration möglich ist, kann die Selektionsmaske sinnvollerweise nur mit den folgenden Werten verknüpft werden:

- Nutzer-ID und
- Gruppen-ID.

Wie in Abbildung 3 dargestellt, lässt sich die Durchsetzung der Aufzeichnungsgranularität in zwei Stufen unterteilen.

Stufe 1 entscheidet, welche Ereignisse grundsätzlich aufgezeichnet werden sollen. An dieser Stelle erfolgt zudem die Selektion hinsichtlich des Ergebniswertes des Systemrufes. Unterscheidungen sind für erfolgreiche, fehlgeschlagene sowie beide Fälle vorgesehen.

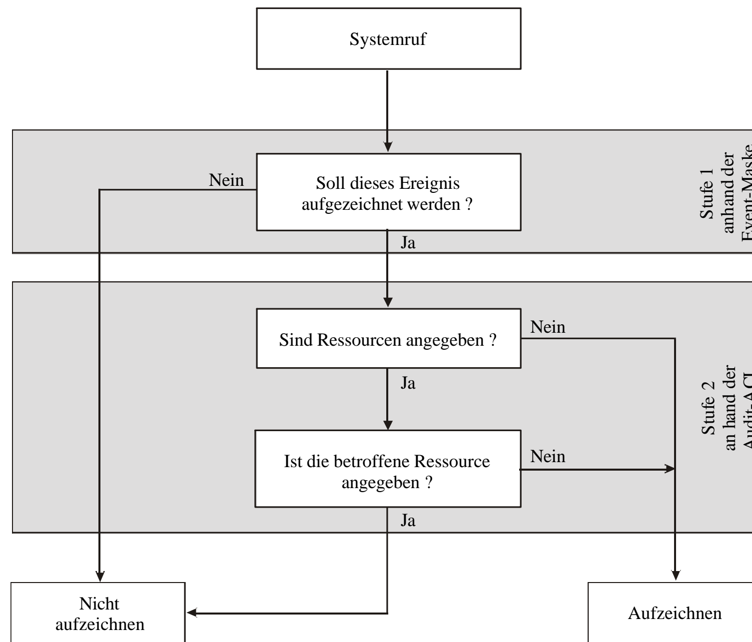

Abbildung 3 : Zweistufige Aufzeichnungsgranulierung

Die zweite Stufe entscheidet, ob die Protokollierung für die betroffene Ressourcen durchgeführt werden soll. Ein Ereignis, welches sich auf ein der Audit-ACL enthaltene Ressource bezieht, wird, wenn dies dort so vermerkt ist (siehe hierzu Abschnitt 4.1.5.2), nicht protokolliert. Ist die Audit-ACL leer bzw. bezieht sich das Event auf keine Ressource, ist die Selektion bereits mit der ersten Stufe abgeschlossen.

Die Angabe der Ressourcen muss, um Zweideutigkeiten auszuschließen, absolut erfolgen. Aus Performanzgründen wurde außerdem auf die Benutzung von regulären Ausdrücken verzichtet.

3.1.3 Sicherheitskonzept

Da das Audit-System selbst eine sicherheitsrelevante Anwendung darstellt, verfügt LOSA über ein eigenes Sicherheitskonzept. Dazu zählt eine eigene Verwaltung von Zugriffsrechten mittels Capabilities (Fähigkeiten) und die Verwendung von verschiedenen Betriebsmodi.

3.1.3.1 Capabilities

Eines der Hauptangriffsziel auf einem UNIX-System ist das Erlangen von root-Privilegien. Um dem entgegenzuwirken, existieren seit kurzem Projekte, die Zugriffsrechte an sogenannten Capabilities festzumachen. Damit entfällt die Notwendigkeit eines Superusers. Die Autorisation für sicherheitsrelevante Aktionen wird nicht mehr an eine oder an mehrere Nutzer-IDs geknüpft, sondern es existiert eine Datenbank, die darüber Auskunft gibt, ob ein Nutzer die für die Aktion notwendigen Fähigkeiten besitzt.

Für die Initialisierung von LOSA ist ein Nutzer notwendig, der innerhalb des Audit-Systems quasi Superuser-Rechte besitzt. Aufgrund der Tatsache, dass root eines der Hauptangriffsziele ist, wurde in LOSA davon abgesehen, dass der Superuser von LOSA mit root identisch sein muss. Der Audit-Administrator benötigt nur entsprechende UNIX-Privilegien, um die notwendigen System-Skripte ausführen zu können. In vielen Fällen sind diese Privilegien gleichbedeutend mit root-Rechten. Um diese Einschränkung umgehen zu können, bedarf es einiger Veränderungen an den Zugriffrechten einzelner Verzeichnisse bzw. System-Skripten.

Da der Capability-Mechismus für Linux noch nicht vollständig implementiert ist und noch nicht Teil des offiziellen Kernels ist, wurde für LOSA ein eigener Capability-Mechanismus geschaffen, der

leicht in das Kernel-Capability-Konzept integriert werden kann. LOSA unterstützt folgende drei Capabilities:

- **`admin`**, erlaubt die Administration des Audit-Systems,
- **`setaid`**, erlaubt das Setzen der Audit-ID und
- **`msg`**, erlaubt das Generieren von Audit-Ereignissen.

Der letzte Punkt ist nicht mit dem eigentlichen Audit zu verwechseln. Audit-Events, die vom Kernel generiert werden, sind unabhängig von Capabilities. Die Capability **`msg`** wird nur von Programmen benötigt, die selbst Kernel-unabhängige Audit-Events generieren wollen. Z.B. benötigt das login-Programm die Capability **`msg`**, damit es das Anmelden eines Nutzer signalisieren kann. Damit soll verhindert werden, dass ein Angreifer falsche bzw. Unmengen von Audit-Events generiert kann, um seine eigentlichen Absichten zu verschleiern.

3.1.3.2 Verschiedene Betriebsmodi

Der zweite Punkt zur Sicherung des Audit-Systems ist die Verwendung von verschiedenen Betriebsmodi. Die Einstellung des Audit-Betriebsmodus erfolgt beim Booten mittels eines Kernel-Parameters oder zur Laufzeit mittels des Kontrollprogramms. LOSA kann in einem der folgenden drei Modi betrieben werden.

`disabled`

Das Audit-System ist deaktiviert, d.h. in diesem Modus kann der Audit-Dämon nicht gestartet werden und es können keine Rekonfigurierungen vorgenommen werden. Mittels des Kontrollprogramms kann in einen der beiden anderen Modi gewechselt werden.

`normal`

In diesem Modus kann der Audit-Dämon gestartet werden und die Protokollierung des Systems ist im vollen Umfang möglich. Allerdings wird keine Sicherheitsüberprüfung vorgenommen. Die zuvor erwähnten Capabilities können zwar gesetzt werden, sie werden jedoch vom System nicht überprüft. Dieser Modus dient zur Konfiguration des Audit-System. In der Praxis zieht sich die Konfiguration des Audit-System über einen längeren Zeitraum hin. Dabei werden immer wieder Parameter des Systems verändert, die im späteren Betrieb nicht mehr angefasst werden und auch nicht mehr angefasst werden sollten. (Dies betrifft vor allen Dingen die Vergabe von Audit-IDs und das Setzen von Capabilities.) Im **`normal`**-Modus ist dies zur Vereinfachung der Konfigurierung ungeschützt möglich.

Wird der Rechner ohne die Angabe es Audit-Betriebsmodus gestartet, befindet sich das System automatisch im Modus **`normal`**.

`secure`

Der **`secure`**-Modus ist der eigentliche Betriebsmodus des Audit-Systems. Hier werden die Capabilities überprüft und die Konfiguration des Audit-Systems ist nur noch eingeschränkt möglich. Außerdem kann der Modus nur durch ein Neustart des Systems gewechselt werden. Dieser Modus sollte nicht mittels des Kernel-Parameters aktiviert werden, da sonst eine Konfiguration des Systems unmöglich ist. Im Normalfall wird der Rechner im Modus **`normal`** gestartet und in einem der Boot-Skripte erfolgt die Konfigurierung des Audit-Systems und der Wechsel in den **`secure`**-Modus.

3.1.4 Konfigurierungsmöglichkeiten

Die Konfigurierung des Audit-Systems kann mittels des Kontrollprogramms zur Laufzeit oder mittels der Konfigurationsdateien:

- `/etc/security/audit.conf`,
- `/etc/security/auditd.conf`,

- `/etc/security/audit_aid.conf`,
- `/etc/security/audit_evt.conf` und
- `/etc/security/audit_acl.conf`.

erfolgen.

audit.conf

Die Datei `audit.conf` enthält die allgemeine Konfiguration des Audit-Systems. Sie wird beim Starten des Rechners von einem der init-Skripte ausgewertet (bei einer RedHat-Standardinstallation ist dies `/etc/rc.d/init.d/audit`). Zur allgemeinen Konfiguration zählt:

- die Entscheidung zur Konfigurierung des Audit-Systems,
- das Starten des Audit-Dämon,
- die Festlegung des Betriebsmodus,
- die Ausführung des Applikationen-Wrappers und
- die Vergabe von Capabilities.

In der Konfigurationsdatei werden dazu folgende Schlüsselwörter verwendet:

- `boot`, bestimmt in wie weit das Audit-System gestartet wird
  - `disabled`, das init-Skript führt keine Aktion aus,
  - `config`, das Audit-System wird konfiguriert, aber der Audit-Dämon wird nicht gestartet
  - `run`, startet zusätzlich den Audit-Dämon,
- `mod`, legt den finalen Betriebsmodus fest,
- `app_wr`, bestimmt, wann die Wrapper-Skripte erzeugt werden sollen (siehe 4.5.2),
  - `script`, bei jedem Start werden die Wrapper-Skripte neu erzeugt,
  - `manual`, das Erzeugen der Skripte wird ausgelassen,
- `cap`, dient dem Einfügen von Capabilities
  - `aid`, fügt für die angegebene Audit-ID die angegebene Capabilities ein,
  - `prog`, fügt für das angegebene Programm die angegebene Capabilities ein.

Jede Regel muss in einer Zeile stehen. Leerzeilen und Zeilen, die mit ‘#‘ beginnen, werden ignoriert. Kommt ein Schlüsselwort zweimal in der Konfigurationsdatei vor, wird die letzte Regeln verwendet. Bei Regeln, die sich über mehr als eine Zeile erstrecken, muss dies durch ein ‘\‘ am Ende der Zeile gekennzeichnet werden. Ist die nächste Zeile eine Leer- oder eine Kommentarzeile, wird die übernächste Zeile hinzugezogen.

Obwohl die Capabilities als eine Regel betrachtet werden, muss jede einzelne Capability in einer Zeile stehen. Die einzelnen Zeilen müssen durch einen Backslash getrennt werden. Geschieht dies nicht wird die neue Zeile als einee neue Regel betrachtet und nicht als Capability aufgenommen. Stehen zwei Capabilities in einer Zeile, wird nur die erste von beiden eingefügt.

auditd.conf

Die Konfigurierung des Audit-Dämon erfolgt mittels der Datei `auditd.conf`. Die Standardinstallation enthält eine Beispielkonfiguration, die für die meisten Systeme verwendet werden kann. In der Konfigurationsdatei können folgende Parameter spezifiziert werden:

- `trail`, legt das Verzeichnis für die Audit-Dateien und den minimal zu verbleibenden Speicherplatz fest,
- `limit`, gibt die maximale Größe einer Audit-Datei an,
- `pipe`, spezifiziert die erste Fifo-Datei für das Steuerungsprogramm (siehe Abschnitt 4.3),

- `connect`, bestimmt die Datei zur Ablage der Audit-Pid und der durch den Parameter `pipe` festgelegten Fifo-Datei,
- `buffer`, unterteilt sich in:
  - `record`, gibt die Anzahl der im Kernel zwischengespeicherten Audit-Records an,
  - `exec`, dient der Verwaltung des Prozess-Environment-Puffers,
- `shutdown`, wird ausgeführt, wenn kein Platz mehr für Audit-Daten vorhanden ist,
- `warn`, wird beim Auftreten eines Fehlers oder einer Warnung ausgeführt.

Die Angabe der Datei- und Speicherplatzgrößen kann in der Konfigurationsdatei in Byte oder in Prozent erfolgen. Dabei steht:

- `[Zahl]`, für Byte,
- `k[Zahl]`, für Kilobyte
- `M[Zahl]`, für Megabyte und
- `%[Zahl]`, für Prozent vom noch verfügbaren Speicherplatz.

Die Prozentangabe kann nur bei der Spezifizierung des zu verbleibenden Speicherplatzes im Trail-Verzeichnis verwendet werden. Der genaue Wert wird vom Audit-Dämon bei der ersten Verwendung des Verzeichnisses berechnet und wird anschließend wie eine Angabe über Bytes verwendet.

Die mittels `buffer record` festgelegte Anzahl von Audit-Record muss über 50 liegen, da eine kleinere Anzahl nicht sinnvoll ist und von der Kernel-Funktion zur Initialisierung des Audit-Puffers als fehlerhafte Eingabe bewertet wird. Der gewählte Werte sollte jedoch nicht zu groß sein, da der Speicherplatz für die Records im gesamten allokiert und damit vom Nutzerspeicher abgezogen wird. Bei den bisherigen Tests hat sich ein Wert von 500 Records als gut erwiesen und kann im allgemeinen so übernommen werden.

Die Angabe des Prozess-Environment-Puffers ist nicht zwingend erforderlich. Wird kein Puffer zur Verfügung gestellt wird, wird der Systemruf `execve(..)` ohne Programm-Environment aufgezeichnet. Die Eigenschaften des exec-Puffers werden mittels der beiden Parameter `order` und `size` beschrieben. Mittels des Schlüsselwortes `order` wird die Größe der Speichersegmente innerhalb eines Puffer-Elementes bestimmt. Die Zahl muss zwischen 0 und 5 liegen. Dabei berechnet sich die Größe des Speichersegmentes wie folgt:

```
mem_size := 4096 * 2 ^ order
```

Bei dem Wert 0 werden initial immer 4096 Byte allokiert, sollte der Platz für das Programm-Environment nicht ausreichen, wird eine zweiter Speicherbereich von 4096 Byte allokiert. Da das Speicherallokieren und das Verketten der einzelnen Speicherbereich wenig performat ist, kann die Größe des Speicherbereiches mittels `order` variiert werden. Nach jedem Speichern des Audit-Puffers werden die Elemente des exec-Puffer freigegeben. Mittels des Parameters `size` kann dies für ein Anzahl von Records unterbunden werden. Im Betrieb hat sich gezeigt, dass ein Wert von 10 ein gute Größe ist. Die Anzahl der gepufferten Elemente sollte nicht zu groß gewählt werden, da ein einzelnes Element des Puffers bis zu 128 kByte Speicher benötigen kann.

Sollten für die Parameter `shutdown` und `audit` Skripte angegeben sein, ist darauf zu achten, dass der Audit-Dämon dies ausführen kann. Ist dies nicht möglich, erfolgt keine Ausgabe eines Fehlers und die Aktion wird nicht ausgeführt. Alternativ zum Skript kann beim Parameter `shutdown` das Schlüsselwort `audit` verwendet werden. In diesem Fall wird nur die Protokollierung deaktiviert und das System läuft normal weiter. Soll auf die Verwendung eines warn-Skriptes verzichtet werden, kann dieses durch das Schlüsselwort `no` ersetzt werden.

audit_aid.conf

Die Datei `audit_aid.conf` enthält die Audit-IDs für Dämon-Prozesse. Die von der Installation mitgelieferte Beispieldatei beinhaltet im wesentlichen alle wichtige Dämon-Prozesse, die bei einer RedHat-Standardinstallation eingerichtet werden. Sollen weitere Prozesse hinzugefügt werden, erfolgt dies mittels:

```
path = [absoluter Pfad der folgenden Programme]
[Programmname_1] [aid_1]
[Programmname_2] [aid_2] leader
...
```

Damit die Änderungen aktiviert werden, muss entweder das Skript `apwr_ctl` aufgerufen oder in der Konfigurationsdatei `audit.conf` der Parameter `app_wr` auf `script` gesetzt werden. Beim Ausführen des Skriptes bzw. beim nächsten Start des Audit-Systems werden die Originaldateien durch eine Datei mit dem Namen `au_[Originalname]` ersetzt und eine Skript mit dem Originalnamen eingefügt. Das eingefügt Skript führt die Originaldatei mittels des Applikation-Wrappers, welcher zuvor die Audit-ID für das zustartende Programm setzt, aus. Erfolgt hinter der Audit-ID die Angabe des Schlüsselwortes `leader` wird das Programm als Session-Leader gestartet. Dies ist bspw. bei allen `getty`-Prozessen notwendig.

Die verwendete Audit-ID sollte, um Verwechselungen zu vermeiden, verschieden von allen Nutzer-IDs sein. Ein negativer Wert sollte dies in jedem Fall gewährleisten.

audit_evt.conf

Um die Konfigurierung des Audit-Moduls zu vereinfachen unterstützt LOSA Event-Klassen. Damit lassen sich einzelne Events zu ganzen Klassen zusammenfassen. Die Event-Klassen werden in der Datei `audit_evt.conf` festgelegt. Dabei sind die Klassennamen frei wählbar, es muss nur beachtet werden, dass nicht mehr als 32 Klassen angelegt werden und folgende Syntax eingehalten wird:

```
Klassenname : Klassen-ID : Event, Event, ...
```

Die Namen der Events sind der Datei `/usr/include/linux/audit_event.h` zu entnehmen. Jedes Event, welches in der Datei `audit_evt.conf` nicht auftaucht, wird der Klasse 0 zugeordnet.

audit_acl.conf

Mittels der Datei `audit_acl.conf` wird die Aufzeichnungsgranularität konfiguriert. Die Konfigurierung erfolgt mittels zwei Arten von Regeln:

- Event-Regeln
- Ressourcen-Regeln

Event-Regeln aktivieren die Aufzeichnung eines Events für einen bestimmten Nutzer. Ressourcen-Regeln deaktivieren die Aufzeichnung eines Events für einen bestimmten Nutzer und eine bestimmte Ressource. Eine Ressourcen-Regel ist immer eine Verfeinerung einer Event-Regel. Man beachte, dass sich die Regeln wiedersprechen können. Wurde bspw. die Protokollierung des Events `AUE_EXECVE(3)` für den Nutzer 101 deaktiviert, sind alle Ressourcen-Regeln, die sich auf das Event 3 und den Nutzer 101 beziehen überflüssig. Sie werden eingefügt, jedoch bei der Generierung des Audit-Events nicht ausgewertet.

Die genaue Grammatik der Regeln ist im Anhang C.3 beschrieben.

3.2 Bestandteile von LOSA

In Abbildung 2 auf Seite 14 sind die einzelnen Komponenten von LOSA und ihre Beziehungen zueinander dargestellt. Das gesamte Audit-System lässt sich in folgende Subsysteme unterteilen:

- die Kernel-Erweiterungen,
- die Audit-Bibliothek,
- der Ereignis-Konfigurations-Optimierer,
- der Audit-Dämon,
- das Audit-Steuerungsprogramm und
- den Trail-Viewer.

Bei den ersten 6 Subsystemen handelt es sich um fundamentale Bestandteile von LOSA, d.h. wird eine dieser Komponenten nicht verwendet, ist es nicht möglich Audit-Daten zu erzeugen. Der Trail-Viewer ist nur zur Betrachtung der Audit-Daten notwendig, ist aber nicht zu deren Generierung erforderlich. Die Audit-Daten werden in einer speziell formatierten Binär-Datei abgelegt. Aus diesem Grund ist ein Auslesen der Daten nicht ohne eine zusätzliche Applikation möglich.

Im folgenden wird die Bedeutung und die Funktionsweise der ersten fünf Komponenten kurz vorgestellt. Ein genauere Beschreibung hinsichtlich Aufbau und Implementierung der Komponenten erfolgt im Kapitel 4. Die Erläuterung des Trail-Viewer erfolgt am Ende dieses Abschnittes. Auf eine genaue Beschreibung der Implementierung des Trail-Viewers wird verzichtet, da er nicht Kern-Bestandteil von LOSA ist.

3.2.1 Kernel-Erweiterung

Den größten Teil von LOSA macht die Erweiterung des Betriebssystem-Kernels von Linux aus. Im Kernel erfolgt die eigentliche Generierung der Audit-Daten, die Verwaltung der gerade erzeugten Records, die Durchsetzung der Aufzeichnungsgranularität und die Einhaltung der Capabilities. Die Verbindung des Audit-Systems mit dem Standard-Kernel erfolgt an zwei Schnittstellen:

1. Erweiterung der aufzuzeichnenden Systemrufe,
2. Erweiterung der Prozess-Struktur.

Jeder Systemruf, der durch das Audit-System aufgezeichnet wird, wurde dahingehend erweitert, dass er die erste Stufe der Durchsetzung der Aufzeichnungsgranularität initiiert und anschließend seine zugehörige Audit-Funktion in der Kernel-Erweiterung aufruft. Für jeden Systemruf existiert eine extra Audit-Funktion. Dies verhindert die übermäßige Verwendung von Vergleichsoperationen und senkt den erzeugten Overhead des Audit-Systems. In der Audit-Funktion selbst erfolgt die Überprüfung der Ressourcen-Regeln und die Generierung des Audit-Records.

Die zweite Schnittstelle dient der Einbringung von fundamentalen Audit-spezifischen Verwaltungsdaten. Linux biete von sich aus keine Datenstrukturen, welche für ein Audit notwendig sind. Hierzu zählt die Audit-ID, die Session-ID, der Herkunftsrechner sowie die Selektionsmasken. Diese Informationen wurden im Rahmen des LOSA-Projektes in die Prozess-Struktur eingefügt.

Im Gegensatz zum BSM von Solaris ist LOSA nicht als Modul implementiert, d.h. soll die Audit-Funktionalität aus dem Kernel entfernt werden, muss dieser neu übersetzt werden. Allerdings sind alle Audit-Operationen, außer den Stubs in den Systemrufen, in einer Objekt-Datei enthalten, so dass LOSA ohne große Modifikationen als Audit-Modul umgesetzt werden kann. Im folgenden wird die Kernel-Erweiterung auch als Audit-Modul bezeichnet. Die Funktionen des Moduls erbringen die eigentliche Audit-Funktionalität im Kernel. Sie werden erst beim Aktivieren der Protokollierung aktiviert. Ist die Protokollierung nicht aktiviert, erzeugen die anderen Erweiterungen nur einen geringen Overhead. Alle aufwendigeren Operationen erfolgen im Audit-Modul.

3.2.2 Audit-Bibliothek

Die Audit-Bibliothek ist die allgemeine Schnittstelle zum Audit-Modul, über welche sowohl die Bestandteile des Audit-Systems als auch andere Applikationen mit dem Audit-Modul im Kernel interagieren können. Nach Außen verfügt das Kernel-Modul nur über einen Systemruf[5]. Dieser dient als Interface, über welches alle andere Audit-Funktionen mittels verschiedener Parameter aufgerufen werden können.

Die Audit-Bibliothek stellt für jede Kernel-Funktion eine Bibliotheks-Funktion zur Verfügung, welche die Übertragung der Parameter übernimmt. Damit bleibt dem Nutzer die Verwendung nur eines Systemrufes verborgen und er muss sich nicht um die Übersetzung der Argumente kümmern. Das gleiche Konzept wird bei Linux unter anderem auch für alle Socket- und NFS-Funktionen verwendet.

Abgesehen von der Übersetzung der Argumente der Audit-Funktionen, werden durch die Funktionen die Audit-Bibliothek dem Kernel, die von ihm zur Durchsetzung der Aufzeichnungsgranularität benötigen Informationen, übergeben werden. Für jeden Nutzer, der sich mittels des im Anhang B.3.1 beschriebenen login-Programms an das System und damit auch an das „Audit-Subsystem" anmeldet, wird die ihm gehörige Ereignismaske (Preselektionsmaske) und Ressourcen-Regeln (Elemente der Audit-ACL) in die Strukturen des Kernels übertragen. Die nötigen Informationen werden vom Audit-Ereignis-Konfigurations-Optimierer in der Datei `audit_acl.db` im Verzeichnis `/etc` hinterlegt und von der Audit-Bibliothek aus dieser Datei ausgelesen.

Um Anwendungen, die nicht direkt zum Audit-Modul gehören, eine möglichst leichte Anbindungen zu schaffen, enthält die Audit-Bibliothek auch alle notwendigen Funktionen zum Auslesen des Audit-Trails. Somit können Applikationen wie der Trail-Viewer unabhängig vom eigentlichen Trail-Format implementiert werden. Eine genaue Beschreibung der verwendeten Strukturen und angebotenen Schnittstellen ist im Anhang C enthalten.

3.2.3 Audit-Ereignis-Konfigurations-Optimierer

Die Konfigurierung der Aufzeichnungsgranularität erfolgt mittels Konfigurationsregel, die in die Konfigurationsdatei

`/etc/security/audit_acl.conf`

eintragen oder mittels des Steuerungsprogramms **aca** an das Audit-Modul übergeben werden können. Der Audit-Ereignis-Konfigurations-Optimierer erstellt aus der Konfigurationsdatei mittels der gdbm-Bibliothek *(GNU database indexing library)* eine Datenbank, welche in der Datei

`/etc/audit_acl.db`

abgelegt wird. Diese Datei enthält die Konfigurationsregeln in binärer Form und kann mittels der Funktionen der gdbm-Bibliothek effizient ausgelesen werden. Die Datei `/etc/audit_acl.db` wird bisher einzig von der Audit-Bibliothek verwendet. Ist diese Datei nicht vorhanden, ist die Audit-Bibliothek nicht in der Lage, die Konfigurationsregeln an das Audit-Modul zu übertragen.

Die Daten, die für erste Stufe der Auswahl der aufzuzeichnenden Audit-Events erforderlich sind, werden in der Preselektionsmaske der Task-Struktur als Bit-Muster abgelegt. Für die zweite Stufe wird im Kernel ein umfangreiche Datenstruktur angelegt, die bei jeder Überprüfung durchsucht werden muss. Im Vorfeld wurde bereits der Mechanismus der ACL erläutert. Die Ressourcenregeln von LOSA werden ähnlich verwaltet. Für jede Ressource mit eigenen Regeln wird ein Element (ACE) in der ACL angelegt. Dieses Element enthält ein Liste mit Event-Masken und Nutzeridentifikatoren, die darüber Auskunft gegeben, ob diese Ressource überwacht werden soll oder nicht (siehe Abschnitt 4.1.3). Da Linux die ACL noch nicht auf der Dateisystemebene unterstützt, wird die Audit-ACL

[5] Die Anzahl der maximal möglichen Systemrufe ist begrenzt und sollte aus Performanzgründen so klein wie möglich gehalten werden.

dynamisch im Speicher gehalten. Im Moment ist es nur möglich Ressourcen hinzuzufügen. Damit der Speicherbedarf für die ACL nicht zu groß wird, muss die Anzahl der Ressourcenregel hinsichtlich Anzahl und Art vom Administrator wohl überlegt sein.

3.2.3.1 Startoptionen

Das Erzeugen der ACL-Datenbank erfolgt mittels **`aclctl [Option] [acl-Datei]`**, wobei folgende Optionen verwendet werden können:

- **`d`**: überschreibt eine alte ACL-Datenbank. Wird diese Option nicht angegeben, obwohl die Zieldatei bereits existiert, wird das Übersetzen abgebrochen.
- **`f [filename]`**: erlaubt die Angabe einer alternativen Zieldatei und
- **`c [filename]`**: verwendet `filename` als Konfigurationsdatei

Wird keine Konfigurationsdatei angegeben, wird standardmäßig die Datei `audit_acl.conf` aus dem Verzeichnis `/etc/security` verwendet.

Von der Audit-Bibliothek wird nur die Datei `audit_acl.db` im Verzeichnis `/etc` verwendet. Wurde mittels der Option –f eine andere Datei erzeugt, wird diese von der Bibliothek **nicht** verwendet.

3.2.4 Audit-Dämon

Der Audit-Dämon ist neben dem Audit-Modul die zweite zentrale Komponente von LOSA. Beide Komponenten müssen zum Generieren von Audit-Daten installiert und gestartet werden. Der Audit-Dämon übernimmt alle Operationen, die nicht im Kernel implementiert werden können bzw. dort nicht implementiert werden sollten.

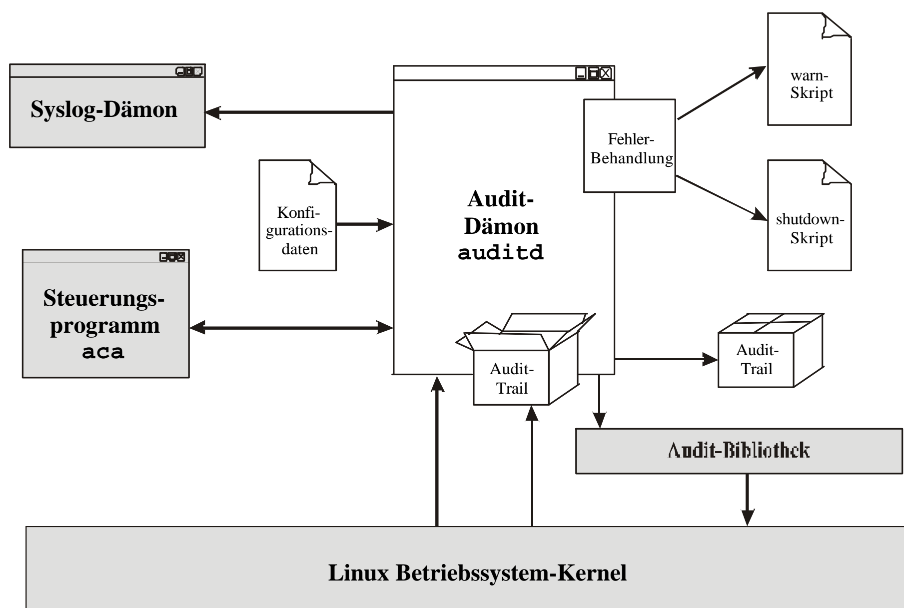

Abbildung 4 : Detailübersicht des Audit-Dämon

In einem UNIX-System ist es eher untypisch, dass der Kernel von sich aus Aktionen ausführt. Die Initiierung einer Aktion wird in der Regel immer von einem Nutzerprozess übernommen[6]. Dies hat den Vorteil, dass alle Operationen im Kernel an einen Nutzerprozess gebunden werden können und

[6] Mittlerweile existieren auch Ausnahmen, so wird die neue Variante des NFS-Dämon komplett in den Kernel integriert. Zwar verwendet man auch hier einen Dämon der die Aktion initiiert, allerdings ruft dieser nur eine Kernel-Funktion auf, die nie wieder zurückkehrt.

damit so wenig wie möglich Funktionalität im Kernel abgearbeitet wird. Dies geschieht nicht nur aus Performanzgründen, sondern ist bei der Behandlung von Fehler und undefinierten Aktionen die einzig mögliche Implementierung. Oftmals ist es dann in einem Fehlerfall ausreichend, dass der betroffene Prozess beendet wird. Das Gesamtsystem kann unbeschadet weiterlaufen.

Bei LOSA ist der Audit-Dämon für die Speicherung der im Kernel erzeugten Audit-Daten, für die Verwaltung der Audit-Trails sowie für die Administration der Audit-Puffer im Kernel verantwortlich. Der Zeitpunkt, wann die Audit-Daten gespeichert werden, wird von Kernel angegeben. Die Speicheroperation selbst, wird vom Audit-Dämon initiiert und dann wiederum komplett im Kernel ausgeführt. Dies hat den Vorteil, dass sich die Audit-Daten niemals im Nutzersegment des Speichers befinden und damit immer vor einem Angreifer bzw. einem unauthorisiertem Zugriff geschützt sind.

Die Administration der Audit-Puffer ist nur zur Performanzsteigerung und zur Anpassung an die jeweilige Hardware von Bedeutung. Eine genauere Beschreibung erfolgt im Abschnitt 4.1.5.

Eine Rekonfigurierungen des Audit-Dämon zur Laufzeit muss aufgrund der Tatsache, dass ein Dämon-Prozess kein Kontrollterminal besitzt, über eines zusätzliches Programm erfolgen. Im Normalfall geschieht dies mittels des im Abschnitt 3.2.5 vorgestellten Steuerungsprogramms. Die Kommunikation zwischen diesen beiden Programmen erfolgt über Signale und mittels benannter Pipes. Eine genaue Beschreibung des Verbindungsaufbaus und der Kommunikation erfolgt im Abschnitt 4.2.

3.2.4.1 Standardausgabe über den syslog-Dämon

Da der Audit-Dämon, wie jeder Dämon-Prozess kein Kontrollterminal besitzt, muss die Ausgabe über einen anderen Weg erfolgen. Der Audit-Dämon nutzt standardmäßig den im Abschnitt 2.2.1 beschriebenen Syslog-Mechanismus zur Ausgabe von Fehlermeldungen und zur Reaktion auf Konfigurierungsoperationen. Um die Informationen des Audit-Systems in der syslog-Datei, i.a. `/etc/log/messages`, kenntlich zu machen, wird jeder Eintrag mit dem Schlüsselwort `audit` versehen. Im normalen Betrieb produziert der Audit-Dämon nur wenige Ausgabe.

Folgende Informationen werden mittels des Syslog-Dämon abgespeichert:

- starten und beenden des Audit-Dämon,
- initiale Größe der Audit-Buffers,
- Mitteilung, dass eine Verzeichnis sein Limit erreicht hat,
- Ausgabe von Fehlermeldungen und
- die Ausgabe des finalen Status einer Regkonfigurierung.

Im Normalfall sollten die Punkte 1 und 2 nur beim Booten und beim Herunterfahren des Systems erscheinen, anderenfalls ist dies ein Indiz dafür, dass der Audit-Dämon womöglich unautorisiert manipuliert oder deaktiviert wurde. Das Erreichen von Limits ist im allgemeinen zu vermeiden, da es unter Umständen passieren kann, dass kein Speicherplatz mehr zur Verfügung steht und andere System-beeinflussende Aktionen eingeleitet werden müssen (siehe Abschnitt 3.2.4.4). Der 4. Punkt, die Ausgabe von Fehlermeldung, sollte verständlicherweise möglichst selten auftreten. Auch an dieser Stelle können, mittels des weiter unter beschriebenen Skriptes, Aktionen eingeleitet werden, die über eine einfache Nachricht an den Syslog-Dämon hinausgehen. Unter dem letzten Punkt fallen alle Nachrichten, die in Folge einer Rekonfiguration des Audit-Systems zur Laufzeit, auftreten.

Da die Anpassung des Audit-Systems an die jeweilige Systemumgebung einen längere Zeit dauern kann und an dieser Stelle mitunter eine Vielzahl von Rekonfigurierungen notwendig sind, ist neben der Ausgabe durch den Syslog-Dämon auch ein Ausgabe zur Shell des in Abschnitt 3.2.5 beschriebenen Steuerungsprogramms möglich. Hierzu muss zum Audit-Dämon mittels des Steuerungsprogramms eine zusätzliche Pipe geöffnet werden und mit Hilfe des Kommandos `getout` die Ausgabe aktiviert werden (vgl. Abschnitt 4.2). Anschließend werden alle Nachrichten sowohl an den Syslog-Dämon geschickt und auf der Shell des Steuerungsprogramms ausgegeben.

3.2.4.2 Start-Optionen

Dem Audit-Dämon können zwei verschiedene Optionen übergeben werden:

- `-d`, der Audit-Dämon läuft nicht als Hintergrundprozess,
- `-f [filename]`, dient der Angabe einer alternativen Startkonfiguration.

Läuft der Audit-Dämon nicht als Hintergrundprozess, sondern als normales Programm mit einem Ein- und einem Ausgabeterminal, werden alle Ausgaben, die sonst an den Syslog-Dämon übergeben werden, auch im Terminal ausgegeben. Außerdem kann der Audit-Dämon in diesem Modus mittels `ctrl-d` beendet werden.

Wird keine Konfigurationsdatei angegeben, versucht der Audit-Dämon eine Datei namens `auditd.conf` im Verzeichnis `/etc/security` einzulesen und zu interpretieren. Ist diese Datei nicht vorhanden und keine alternative Datei angegeben, schlägt das Starten des Audit-Dämons fehl.

3.2.4.3 Trail-Verwaltung

Die Audit-Trails müssen lokal auf dem Rechner abgespeichert werden. Es sei denn, das Trail-Verzeichnisse verweist auf ein via NFS eingebundenes Verzeichnis. Dies wäre bspw. bei der Verwendung eines Log-Servers, welcher die Trail-Verzeichnisse an seine Clients mittels NFS exportiert, möglich. Mittels einer solchen Konfiguration können alle Audit-Trails zentral auf dem Log-Server hinterlegt und ausgewertet werden.

Die Benennung der Audit-Trails erfolgt durch den Audit-Dämon nach einem vorgegebenen Schema. Dabei setzt sich der Name aus der Startzeit, der Endzeit der Abspeicherung und des Rechnernamens zusammen. Als Format wird folgendes Schema verwendet:

```
Startzeit.Endzeit.Rechnername
```

Das Format der Zeitangabe ist so gewählt, dass sich die einzelnen Trails lexikographisch vergleichen lassen. Dadurch lässt sich schon anhand der Trail-Namen die chronologische Reihenfolge der Datengenerierung rekonstruieren. Die Startzeit ist der Zeitpunkt, zu welchem der Trail geöffnet wurde und die Endzeit, wann dieser wieder geschlossen wurde. Solange der Trail verwendet wird, ist die Endzeit mit `not_terminated` belegt.

```
19991129102058.not_terminated.nikita
19991129102058.19991129102154.nikita
```

Die beiden Dateinamen im obigen Beispiel verweisen auf einen Audit-Trail, der am 29.11.1999 um 10.20.58 geöffnet und am selben Tag um 10.21.54 geschlossen wurde. Die Audit-Daten wurden auf dem Rechner `nikita` erzeugt. Konnte ein Audit-Trail nicht korrekt geschlossen werden, wird die Endzeit nicht gesetzt und der Trail damit als fehlerhaft bzw. unvollständig gekennzeichnet.

Unter der Verwaltung der Audit-Trails ist vor allem auch die Einhaltung der Limits zu verstehen. Dem Audit-Dämon können folgende Limits übergeben werden:

- der minimal zuverbleibende Speicherplatz im Trail-Verzeichnis und
- die maximale Größe eines Audit-Trails.

Ist die maximale Größe eines Audit-Trails erreicht, wird dieser geschlossen und ein neuer Trail im gleichen Verzeichnis angelegt. Ist das Limit in einem Audit-Verzeichnis erreicht, werden die Audit-Daten ins nächste Trail-Verzeichnis gespeichert. Sind alle dem Audit-Dämon bekannten Verzeichnisse bis zu ihren Limits gefüllt, wird das Auditing deaktiviert und die *shutdown*-Aktion ausgeführt.

3.2.4.4 Fehlerbehandlung

Für eine sichere *shutdown*-Aktion sollte im *shutdown*-Skript in Skript oder ein Programm angegeben werden, welches das System herunterfährt oder in einen anderen sicheren Zustand überführt. Damit wird die Anzahl der nicht protokollierten Aktionen so gering wie möglich gehalten. Wurde kein Skript angeben, wird lediglich der Audit-Dämon beendet. Neben dem *shutdown*-Skript kann dem Audit-Dämon ein *warn*-Skript übergeben werden. Im Gegensatz zum *shutdown*-Skript welches ohne Argumente ausgeführt wird, wird dem *warn*-Skript einer der folgenden Parameter übergeben:

- **`limit`** : ein Verzeichnis wurde bis zum Limit gefüllt, der zweite Parameter enthält den Namen des betroffenen Verzeichnisses,
- **`all`** : alle Verzeichnisse wurden bis zu ihren Limits gefüllt,
- **`perm`** : der Zugriff auf das Audit-System wurde verweigert,
- **`busy`** : den Audit-Dämon wurde ein zweites mal gestartet und
- **`nostart`** : der Audit-Dämon wurde aufgrund eines unbekannten Fehlers beendet.

Das Standard-*warn*-Skript schickt in allen Fällen eine E-Mail an den Systemadministrator. Die Aktionen des Skriptes sind nicht an das Audit-System gebunden, es steht dem Audit-Administrator frei, welche Aktion in den jeweiligen Fällen ausgeführt werden sollen.

3.2.5 Audit-Konfigurationsinterface

Das Audit-Konfigurationsinterface ist die Managementkomponente von LOSA. Während der Audit-Dämon für die Verwaltung der Audit-Trails und aller damit verbundenen Bestandteile im Kernel verantwortlich ist, dient das Audit-Konfigurationsinterface der Administrierung des Audit-Dämon zur Laufzeit und der Konfigurierung aller Dämon-unabhängigen Kernel-Komponenten. Hierzu zählt die Verwaltung der Capabilities, die Konfigurierung der Selektionsregeln und die Festlegung des Betriebsmodus.

Das Programm ist so konzipiert, dass damit das Audit-System zur Laufzeit rekonfiguriert als auch zur Bootzeit initialisiert werden kann. Zur Rekonfigurierung des Systems kann das Steuerungsprogramm im interaktiven Modus betrieben werden. Dem Nutzer wird dann eine Shell zur Verfügung gestellt, welche die eingegebenen Kommandos interpretiert und an den Audit-Dämon oder an das Audit-Modul im Kernel überträgt. Zur Initialisierung des Audit-Systems wird das Audit-Konfigurationsinterface in den jeweiligen Skripten aufgerufen. Die Kommandos werden per Parameter direkt übergeben, das Programm versucht dieses auszuführen und beendet sich anschließend. Eine Ausgabe erfolgt in diesem Fall nur beim Auftreten eines Fehlers oder bei der Abfrage von Statusinformationen.

3.2.5.1 Aufrufparameter

Der Aufruf des Konfigurationsinterfaces muss mittels eines Parameters erfolgen, der ein auszuführendes Kommando beinhaltet oder die Ausführung eines Kommandos ermöglicht.

- `-c [command]`, bewirkt, dass `command` ausgeführt wird,
- `-i`, das Programm öffnet die Kommando-Shell (interaktiver Modus),
- `-s`, die Kommandoausführung erfolgt im silent-Mode und
- `-f [connect-Datei]`, erlaubt die Angabe einer alternativen Verbindungsdatei.

Die Parameter können sowohl einzeln oder in Kombination verwendet werden, einzig die Kombination der Parameter `-s` und `-i` ist untersagt. Zur Vereinfachung der Benutzbarkeit in Skripten kann das Steuerungsprogramm mit der Option –s gestartet werden. In diesem Fall wird nur das Ergebnis des Befehls ohne einen zusätzlichen Kommentar ausgegeben.

Wird mittels der Option `-f` eine alternative Verbindungsdatei angegeben, versucht das Konfigurationsinterface mittels dieser Datei die Verbindung zum Audit-Dämon aufzunehmen. Standardmäßig wird die Datei `auditd.pid` im Verzeichnis `/var/run` verwendet. Der

Verbindungsaufbau zum Audit-Dämon erfolgt einmal beim Starten des Programms. Schlägt das Kontaktieren des Audit-Dämon fehl, ist ein erneuter Versuch nur durch einen Neustarten des Programms möglich. In diesem Fall sollte die `connect`-Datei überprüft werden, da diese meist die Hauptfehlerquelle für einen fehlgeschlagenen Verbindungsaufbau ist.

Die Syntax der mittels der Option `–c` übergebenen Kommandos ist identischen mit den in der Shell verwendeten Befehlen. Die Ausgabe des Resultates erfolgt auf die Standardausgabe.

3.2.5.2 Shell-Kommandos

In diesem Abschnitt erfolgt die Erläuterung der Shell-Kommandos, die Grammatik ist im Anhang C.3 ausführlich beschrieben. Die Befehle lassen sich in drei Gruppen unterteilen. Zum einen die Kommandos, die zur Administration der Shell und des Audit-Dämon dienen. Hierzu zählen die Kommandos:

- `next`, signalisiert dem Audit-Dämon einen neuen Trail zu öffnen,
- `reset`, die Audit-Dämon liest die Konfigurationsdatei neu ein,
- `terminate`, die Protokollierung wird deaktiviert,
- `quit`, die Shell wird geschlossen und
- `help`, es erfolgt die Ausgabe aller möglichen Shell-Befehle.

Diese Kommandos können nur in der Shell verwendet werden. Eine Verwendung in den Konfigurationsdateien oder in den Skripten ist wenig sinnvoll, da die Kommandos nur in der Shell des Audit-Konfigurationsinterfaces oder zur Laufzeit des Audit-Dämon eine Bedeutung haben

Alle übrigen Shell-Kommandos dienen der Administrierung des Audit-Systems bzw. dem Abfragen von Statusinformationen. Neben dem Einsatz in der Shell, können die Befehle zum Initialisieren des Audit-Systems verwendet werden. Die Kommandos:

- `setmask`, ermöglicht das Setzen der Preselektionsmaske eines oder mehrerer Prozesse,
- `setacl`, wird zum Setzen der Ressourcen-Regeln eines oder mehrerer Prozesse verwendet,
- `setcap`, dient zur Administrierung der Capabilities,
- `login`, meldet einen neuen Nutzer an das Audit-System an,
- `chmod`, wechselt den Betriebsmodus,
- `getmask`, liefert die Preselektionsmaske eines Prozesses,
- `getadmin`, liefert die Nutzer-ID des Audit-Administrators,
- `getmod`, liefert den aktuellen Betriebsmodus und
- `getcap`, dient dem Abfragen der Capabilities einer Audit-ID,

sind unabhängig von Audit-Dämon. Sie beziehen sich auf statische Informationen im Kernel, die während der gesamten Laufzeit des Systems gesetzt werden können. Während des Betriebsmodus **`disabled`** ist einzig die Ausführung des Kommandos `chmod` und `getmod` möglich. Der Aufruf aller anderen Kommandos liefert in diesem Modus einem Fehler. Wird der Rechner im Audit-Betriebsmodus **`disabled`** gestartet, kann die Protokollierung nicht korrekt initialisiert werden. Damit das Audit-System bei einem späteren Modus-Wechsel zur Laufzeit korrekt funktioniert und die gewünschten Aktionen protokolliert werden, müssen die einzelnen Parameter von Hand gesetzt oder die entsprechenden Skripte nach dem Setzen des Betriebsmodus erneut ausgeführt werden.

Alle Befehle, die zum Setzen von Parametern oder zum Einfügen von Werten dienen, erzeugen bei einer erfolgreichen Ausführen keine Ausgabe. Da die Kommandos im späteren Betrieb nur noch zum Initialisieren benötigt werden und dies in den Start-Skripten der entsprechenden Run-Levels erfolgt, ist einzig Informationen über fehlgeschlagene Aktionen interessant und sinnvoll.

Die letzte Gruppe von Shell-Kommandos dient der Administrierung des Audit-Dämon. Hierzu stehen die Befehle:

- `trail`, gestattet das Hinzufügen eines Trail-Verzeichnisses,
- `limit`, dient dem Setzen der maximalen Trail-Größe,
- `configfile`, ermöglicht die Angabe einer alternativen Konfigurationsdatei,
- `buffer`, dient der Administration des Audit-Puffers im Kernel,
- `warn`, analog `configfile`, in Bezug auf das `warn`-Skript,
- `shutdown`, analog `configfile`, in Bezug auf das `shutdown`-Skript und
- `stat`, in Verbindung mit einem der obigen Befehle wird der aktuelle Wert des zugehörigen Parameters ausgegeben,

zur Verfügung. Die Ausführung eines dieser Kommandos ist nur nach einem erfolgreichen Verbindungsaufbau zum Audit-Dämon möglich. Ist der Audit-Dämon nicht aktiv oder ein Verbindungsaufbau über die angegebene `connect`-Datei nicht möglich, liefert jeder Aufruf eines dieser Kommandos einen Fehler.

3.2.6 Trail-Viewer

Der Trail-Viewer dient als einfaches Analyse-Tool für Audit-Trails, welche mittels LOSA erzeugt wurden. Er ermöglicht neben der Betrachtung der Audit-Daten auch eine selektionsorientiert Suche, welche bei größeren Audit-Trails schnell erforderlich wird. Da die Verarbeitung der Audit-Daten nicht Bestandteil der Aufgabenstellung war, ist der Funktionsumfang des Trail-Viewers auf ein Mindestmaß beschränkt. Die graphische Benutzeroberfläche ist mittels der Skriptsprache Tcl/Tk und dem Zusatzpaket Tix erstellt. Der Trail-Viewer bringt seinen eigenen Interpreter mit, jedoch wird dieser standardmäßig dynamisch erzeugt, so dass die notwendigen Bibliotheken installiert sein müssen.

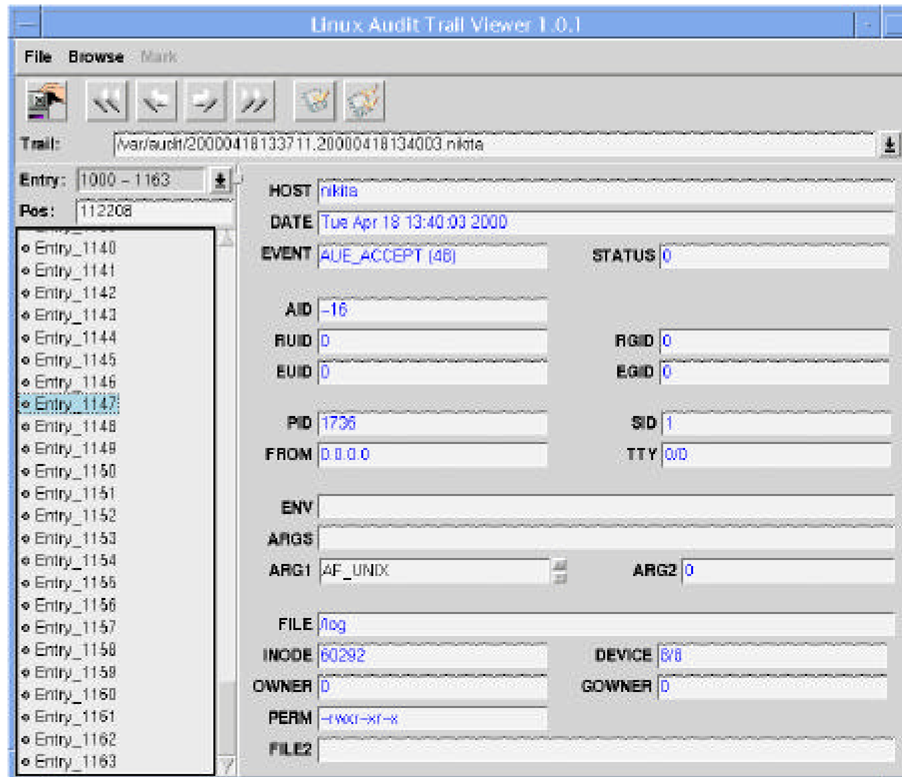

Abbildung 5 : Trail-Viewer Hauptfenster

Abbildung 5 zeigt das Hauptfenster des Trail-Viewers, nachdem ein Audit-Trail geöffnet und ein Event, in diesem Fall eine `accept`-Event, selektiert wurde. Gestartet wird der Trail-Viewer durch das

Kommando **atv [trail-name]**. Die Angabe einer Audit-Datei ist optional. Mittels des Datei-Requesters im Menü `File` kann ebenfalls ein Audit-Trail geöffnet werden. Nachdem ein Trail erfolgreich geöffnet wurde, wird der Dateiname in der ComboBox `Trail` unter der Button-Leiste angezeigt. Wurden im Verlauf der Programmausführung mehrere Dateien geöffnet, enthält die ComboBox ein History der öffneten Dateien. Durch anklicken eines Eintrages kann dieser wieder geöffnet werden. Die untere Hälfte des Hauptfensters gliedert sich in zwei Teile. Die linke Seite dient der Navigation im Audit-Trail, während die rechte Seite den Inhalt des ausgewählten Audit-Records darstellt.

Die Navigation im Trail erfolgt hauptsächlich mittels der großen Listbox im linken unteren Teil. Die Listbox enthält höchstens eintausend Elemente. Mittels der ComboBox `Entry` kann eine anderer „Tausender-Block" ausgewählt werden. Diese etwas komplizierte Navigation wurde aus implementierungstechnischen Gründen notwendig, da Tcl/Tk nicht in der Lage ist mehr als 2000 Einträge in einer Listbox sauber darzustellen. Der Eintrag `Pos` zeigt die aktuellen Byte-Position des aktuellen Eintrages. Die Navigation im Audit-Trail ist ebenfalls mittels der Button-Liste möglich.

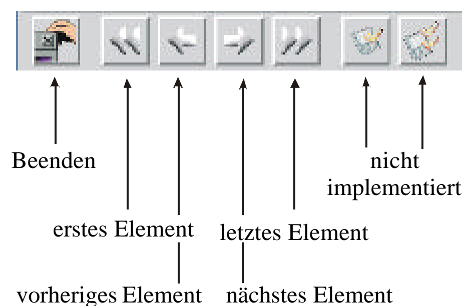

Abbildung 6 : Button-Leiste

Im Menü `Browse` ist Funktionalität der einzelnen Buttons zusätzlich als Menüpunkte verfügbar. Außerdem befindet sich im Menü `Browse` der Eintrag `Search`, welcher das in Abbildung 7 dargestellte Dialogfenster öffnet.

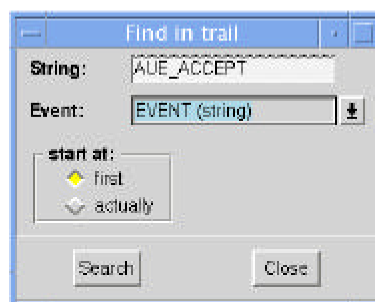

Abbildung 7 : Das Suchdialogfenster

Das Suchdialogfenster ermöglicht eine einfache Suche im Audit-Trail. Dazu kann in der ComboBox `Event` das Suchkriterium ausgewählt werden. Mögliche Suchkriterien sind: Prozess-ID, Audit-ID, Event, als Zeichenkette oder Zahl und die IP-Adresse des `from`-Eintrages. Im Feld `String` wird das gesuchte Element eingetragen. Über die Radiobutton `start at` wird das Element bestimmt, von wo die Suche beginnen soll. `first` startet die Suche beim ersten Element und `current` beim aktuellen Element. Ist keine Element im Hauptfenster selektiert, beginnt die Suche immer beim ersten Element. Wurde ein entsprechendes Element gefunden, wird dies als aktueller Eintrag dargestellt. Bei einer fehlgeschlagenen Suche erfolgt zur Zeit keine Ausgabe.

Kapitel 4

Implementierung von LOSA

Nachdem im Kapitel 3alle Bestandteil von LOSA im einzelnen vorgestellt wurden, erfolgt in diesem Abschnitt die Beschreibung von einigen implementierungstechnischen Feinheiten. Dieser Abschnitt erläutert nicht die komplette Implementierung von LOSA, hierzu ist ein Blick in die Quelle unumgänglich. Ziel ist die Vorstellung der Kernel-Komponenten und die Beschreibung der Schnittstellen.

Das Konzept von LOSA gewährleistet, dass alle Subsysteme von einander entkoppelt sind, so dass unter der Verwendung der entsprechenden Schnittstellen, jedes Modul ausgetauscht bzw. modifiziert werden kann, ohne dass die anderen Teile beeinträchtigt werden. Somit können gezielt einzelne Module an die eigenen Anforderungen angepasst werden.

4.1 Erweiterungen des Betriebssystem-Kernels

Die Kernel-Erweiterung ist der wichtigste und umfangreichste Bestandteil von LOSA. Im Kernel erfolgt die eigentliche Genierung der Audit-Daten, die Verwaltung der gerade generierten Audit-Records und die Durchsetzung der Aufzeichnungsgranularität.

Linux besitzt zwei wesentliche Eigenschaften, die für den Betrieb des Systems und des Audit-Systems fundamental wichtig sind.

1. Wie jedes UNIX-System unterteilt auch Linux den Hauptspeicher des Rechners in einen Kernel- und einen Nutzerbereich. Ein Programm im Nutzer-Modus hat keinen Zugriff auf den Kernel-Bereich, sondern nur auf die Daten in seinem Nutzerbereich. Ein Prozess, der sich im Kernel-Modus befindet, kann über den gesamten Speicher und damit auch über den Nutzerbereich verfügen. Ein Nutzerprozess gelangt durch den Aufruf eines Systemrufs in den Kernel-Modus.

2. Ein Implementierungsrichtlinie von Linux besagt, dass jeder Systemruf bis zum Ende abgearbeitet wird und erst anschließend ein neuer Systemruf ausgeführt werden kann. Ein großer Teil der Kernel-Funktionen ist, damit sie korrekt funktionieren, auf diese Richtlinie angewiesen. Anderenfalls wäre die Verwendung von globalen Variablen innerhalb des Kernels sehr aufwendig und wenig performant. Nahezu jede Schreib- oder Leseoperation müsste mittels Semaphoren oder ähnlichen Mitteln abgesichert werden.

Wie fast alle Kernel-Funktion sind auch die Funktion des Audit-Modus auf diese beiden Eigenschaften angewiesen. Im Audit-System wird die 2.Eigenschaft zusätzlich zur Sicherstellung der chronologischen Reihenfolge der Audit-Records im Audit-Trail verwendet. Jedes Ereignis wird vollständig protokolliert, bis ein anderer Prozess einen neuen Systemruf aufruft und in diesem einen neuen Record anfordern kann[7]. Die Generierung der Audit-Daten erfolgt im Systemruf. Jedes Programm, dass einen Systemruf ausführt, der vom Audit-System überwacht wird, erscheint, so dies vom Audit-Administrator gewünscht wird, im Audit-Trail.

[7] Dies gilt in einem SMP-System nur in soweit, dass die chronologische Reihenfolge von aufeinander aufbauenden Ereignissen korrekt ist. D.h. ein Kindprozesses kann erst Systemrufe ausführen, nachdem die Funktion fork(...) und damit auch die Protokollierung komplett abgearbeitet wurde.

Nicht jeder Systemruf stellt alle für einen Audit-Record nötigen Informationen bereit. In diesen Fällen füllt die Audit-Funktionen den Record mittels globaler Kernel-Variablen, die nur innerhalb des Kernels existieren und niemals nach außen gegeben werden.

Da die Audit-Records sensitive Daten enthalten und einen umfangreichen Einblick in das System gewährt, ist die Sicherung der erzeugten Daten ein wichtiger Aspekt des Audit-Systems. In der Zeit zwischen der Generierung der Audit-Records und ihrer Speicherung im Trail verlassen sie niemals den Kernel-Speicher und bleiben damit vor unbefugten Zugriffen geschützt. Eine sichere Verwaltung der Audit-Trails ist nicht mehr Bestandteil von LOSA und muss von Systemadministrator gewährleistet werden.

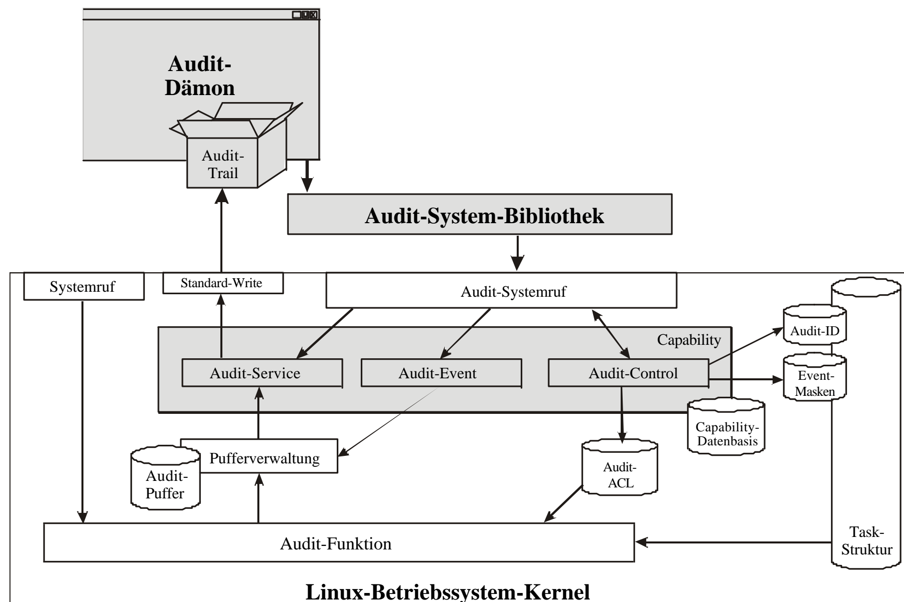

Abbildung 8 : Erweiterung des Betriebssystem-Kernel

Im Hinblick auf Erweiterbarkeit und Austauschbarkeit wurde die Kernel-Erweiterung so konzipiert und implementiert, dass ein möglichst modularer Aufbau entsteht. Zwar war dies nicht so gut möglich, wie bei den globalen Komponenten von LOSA. Jedoch sind die einzelnen Komponenten so implementiert, dass sie zum Großteil in sich abgeschlossen und bis auf wenige Ausnahmen frei von Nebenbedingungen bzw. Seiteneffekten sind.

4.1.1 Der Kernel-Zugangsfunktion `sys_auditcall(...)`

Die Kernel-Architektur von Linux ist in der Lage 256 verschiedene Systemrufe zu verwalten. Zur Zeit werden davon 190 Systemrufe genutzt. Das Audit-Modul kann mittels mehr als 15 verschiedener Funktionen gesteuert werden. Da LOSA nicht das einzige Projekt ist, welches die freien Systemrufe verwendet, ist es nicht möglich jede Funktion über einen separaten Systemruf anzusprechen. Aus diesem Grund werden alle Audit-Funktionen über die allgemeine Audit-Zugangsfunktion `sys_auditcall(...)` aufgerufen.

Die Funktion `sys_auditcall(...)` besitzt zwei Parameter. Der erste Parameter ist ein `int`-Wert, welcher die aufzurufende Funktion spezifiziert. Der zweite Parameter ist ein Zeiger auf ein Array, welches die Argumente der aufzurufenden Funktion enthält. Die möglichen Werte des ersten Parameters sind in der Datei `audit.h` als Defines vorgegeben. Das zweite Argument ist abhängig von der aufzurufenden Funktion. Aus Design-technischen Gründen hat es sich als gut erwiesen, die Aufrufhierarchie mehrstufig zu gestalten. Abbildung 9 zeigt die Aufrufhierarchie aller Funktionen, die über den Systemruf `sys_auditcall(...)` angesprochen werden können.

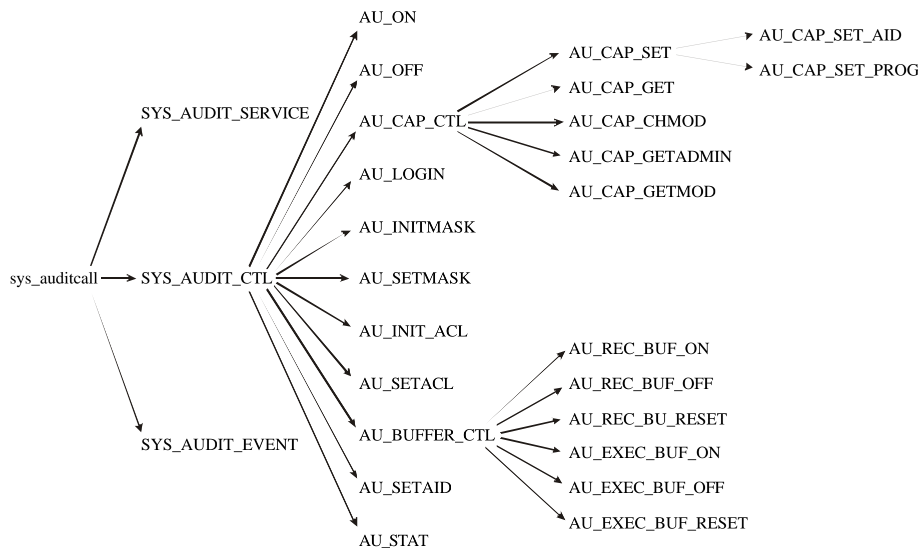

Abbildung 9 : `sys_auditcall(...)` Kommandohierarchie

Da die Programmierung dieser Art von Funktionsaufrufen und Parameterübergaben ungewohnt kompliziert und fehleranfällig ist, stellt die Audit-Bibliothek für jede Kernel-Funktion ein zugehörige Bibliotheksfunktion zur Verfügung, die das Verpacken der Funktionsargumente übernimmt. Es besteht aber nichtsdestotrotz weiterhin die Möglichkeit den Systemruf `sys_auditcall(...)` direkt zu verwenden.

Nicht für jedes in Abbildung 9 aufgeführte Kommando wird von Audit-Bibliothek eine Funktion zur Verfügung gestellt. Einige Kommandos können einzig über die Funktion `audit_ctl(...)` angesprochen werden oder werden nur Kernel-intern verwendet. Im Anhang C.2 sind alle Bibliotheksfunktionen aufgeführt und beschrieben.

4.1.2 Erweiterungen der Prozess-Struktur

Linux ist Prozess-orientiert, d.h. jedes Programm besteht aus mindestens einem Prozess. Die einzelnen Prozesse werden von Linux in einer Prozessliste verwaltet Jedes Element entspricht einem Prozess und wird durch die Prozessstruktur `task_struct` beschrieben. In der Prozessstruktur sind alle relevanten Informationen zu einem Prozess enthalten.

Der erste Prozess **`init`** wird durch eine statische Variable des Kernel, die beim Systemstart initialisiert wird, repräsentiert. Alle weiteren Prozesse werden dynamisch angelegt und in die Prozessliste eingehangen. Das Erzeugen eines neuen Prozesses erfolgt durch die Funktion `do_fork(...)`. Sie erzeugt zunächst eine Kopie des Vater-Prozesses und aktuallisiert die Prozess-ID der Kopie. Ist die Initialisierung abgeschlossen, wird der neuen Prozess durch das Einhängen in die Prozessliste dem System bekannt gemacht. Der neue Prozess ist bis auf die Prozess-ID ein exakte Kopie des Vaterprozesses.

Sollen zusätzliche Elemente in die Prozessstruktur eingefügt werden, so müssen die Änderungen in:

- die Struktur `task_struct`,
- die Initialisierungsmaske und
- die Funktionen `do_fork(...)` und `exit_notify(...)`

eingefügt werden. Für LOSA wurde die Prozessstruktur um die Audit-Struktur und die Preselektionsmaske erweitert.

4.1.2.1 Audit-Struktur

Wie die meisten Unix-Derivate ist Linux ein multi-user, multi-session Betriebssystem, d.h. es erlaubt das gleichzeitige Arbeiten von mehreren Nutzern auf einem Rechner. Nicht immer sind mehrere verschiedene Nutzer gleichzeitig auf einem Rechner angemeldet, aber oft wechselt ein Nutzer temporär seine Identität mittels Programmen wie **`su`** oder **`sudo`**. Während etablierte Systeme wie Solaris eine Identifizierung des Nutzers über temporäre Identitätswechsel hinweg gewährleisten, unterstützt Linux dies nicht. Hat ein Nutzer mittels **`su`** seine Identität gewechselt, ist es unmöglich nachzuvollziehen, wer dieser Nutzer zuvor war.

Da gerade die eindeutige Identifizierung des Initiators Grundlage von Audit ist, wurde im Rahmen von LOSA in die Prozess-Struktur die Audit-Struktur eingefügt.

```
struct audit_struct {
     atomic_t        count;
     int             aid;
     pid_t           sid;
     unsigned long   from;
     gid_t           agid;
     unsigned short  set;
};
```

`count`:
Die Variable `count` zählt die Prozesse, welche die Struktur aktuell verwenden. Normalerweise verfügt jeder Prozess über seine eigene Audit-Struktur, jedoch unterstützt Linux das Konzept der Threads mittels Pseudo-Prozessen. Hierzu werden die einzelnen Threads durch spezielle Prozesse repräsentiert, d.h. hinter jedem erzeugten Thread verbirgt sich immer noch ein Prozess[8]. Threads, die zu einem Prozess gehören, teilen sich den Speicher und den Prozess-Kontext und damit auch die `Audit`-Struktur. Die Aktualisierung der `count`-Variable wird in den Kernel-Funktionen `do_fork(...)` und `exit_notify(...)` durchgeführt.

`aid`:
Hierbei handelt es sich um die eigentliche Audit-ID, die mittels der Funktion `setaid(...)` gesetzt werden kann. Obwohl eine normale Nutzer-ID vom Type `unsigned short` ist, wird für die Audit-ID der Datentyp `int` verwendet[9]. Dies hat den Vorteil, dass eine zusätzliche Anzahl von Nutzer-IDs (negative IDs) zur Verfügung steht. Wodurch verschiedenen Prozesse, die unter der gleichen Nutzer-ID laufen, verschiedene Audit-IDs bekommen können (vgl. Abschnitt 3.1.4).

`sid`:
Die Variable `sid` enthält die Session-ID. Zusammen mit der Audit-ID dient sie der eindeutigen Identifizierung einer Nutzer-Aktion. Die Session-ID wird wie beim BSM von Solaris auf die Prozess-ID des Login-Prozesses gesetzt.

---

[8] Linux unterstützt somit keine echten Kernel-Threads, wie es bei Solaris oder Windows NT der Fall ist. Jedoch werden durch die Standard-C-Bibliothek alle Funktionen zur Unterstützung von Posix-Threads zur Verfügung gestellt und der Kernel sorgt für die Unterstützung des gemeinsamen Prozess-Raumes.

[9] Dies galt bis zur Kernel-Version 2.3.x. Ab der Version 2.4.0 ist die Nutzer-ID vom Typ `int` und kann damit ebenfalls negativ sein. Diese Möglichkeit wird bisher jedoch von fast keinen Programm benutzt, so dass die Vergabe von negativen Nutzer-IDs immer noch zur eindeutigen Identifikation spezieller Prozesse genutzt werden kann.

from:
Neben der Terminal-ID, die von Linux selbst verwaltet wird, wird vor allem bei Netzwerkzugriffen die IP-Adresse des Herkunftsrechners benötigt. Die IP-Adresse wird in der Variable `from` hinterlegt. `from` ist vom Typ `unsigned long` und unterstützt somit keine IPv6-Adressen.

agid:
In der Variable `agid` wird die initiale Gruppen-ID des Nutzers hinterlegt. Die Gruppen-ID wird nicht im Audit-Trail gespeichert. Sie dient lediglich zur Durchsetzung der Aufzeichnungsgranularität. Das Setzen der Variable erfolgt zusammen mit der Session-ID `sid` und Herkunftsrechners `from`.

set:
Da die Audit-Struktur ein sicherheitskritischer Bestandteil von LOSA ist, wurde neben den eigentlichen Daten die `set`-Variable zur Sicherung der Struktur eingefügt. Die Variable `set` kann die Werte 0 und 1 annehmen. Solange die Variable den Wert 0 besitzt, können die anderen Daten der Audit-Struktur verändert werden. Ist der Wert von `set` gleich 1, ist ein Verändern der Daten nicht mehr möglich.

Die Initialisierungsmaske der Struktur ist zusammen mit der Audit-Struktur in der Datei `include/linx/sched.h` definiert. Die Initialisierung der Audit-Struktur erfolgt statisch zusammen mit den anderen Werten der Task-Struktur in der Datei `arch/i386/kernel/init_task.c`.

Die Funktion `do_fork(...)` wurde um die Kopierfunktion `copy_audit(...)` erweitert. Da in der Task-Struktur nur ein Zeiger auf die Audit-Struktur enthalten ist, muss das Kopieren der Struktur von Hand erfolgen. Handelt es sich bei dem neuen Prozess um einen Thread, wird nur die `count`-Variable inkrementiert.

4.1.2.2 Preselektionsmaske

Die Preselektionsmaske (Event-Maske) dient der Durchsetzung der Aufzeichnungsgranularität und ist in einer (32 x 8)-Matrix abgelegt. Implementiert wird die Matrix mit Hilfe eines `unsigned long` – Array mit 8 Elementen. Jedem Event sind zwei Bits zugeordnet, somit können mit der Event-Maske 128 Events verwaltet werden. Jede Zeile (ein Array-Element) enthält die Werte für 16 Events, wobei die Bits pro Event wie folgt belegt sind:

- 00 : keine Auszeichnung,
- 01 : Aufzeichnung der erfolgreichen Systemrufe,
- 10 : Aufzeichnung der fehlgeschlagenen Systemrufe und
- 11 : generelle Aufzeichnung.

Da die Event-Maske ein statisches Element der Prozess-Struktur ist, wird sie von der Funktion `do_fork(...)` automatisch kopiert. Die Event-Maske wird statisch mit 0 initialisiert. Demzufolge werden solange keine Audit-Events aufgezeichnet, bis die Event-Masken der einzelnen Prozesse nicht anderweitig belegt werden.

Das Setzen der Event-Maske erfolgt mittels der Funktionen `audit_setmask(...)` und `audit_init_mask(...)`. Mittels der Funktion `audit_setmask(...)` kann explizit ein Ereignis auf einen bestimmten Wert gesetzt werden. `audit_init_mask(...)` ersetzt die gesamte Event-Maske durch die übergebene Maske.

4.1.3 Audit-ACL

Die ACL, wie sie von Windows NT bekannt ist, enthält normalerweise Informationen für die Zugriffskontrolle des Dateisystems. Für das Standard-Dateisystem von Linux (ext2) existieren zwar ACL-Einträge in den Datenstrukturen, jedoch sind im VFS (virtuelles Dateisystem) keine Funktionen

implementiert, um die Daten verwalten zu können. Somit unterstützt Linux keine ACLs, obwohl die notwendigen Einträge in den Datenstrukturen vorhanden sind[10]. Unter Windows NT werden in den ACEs (access control entry, Elemente der ACL) neben den Zugriffsrechten für das Dateisystem auch die Regeln für die Durchsetzung der Aufzeichnungsgranularität bei Ressourcen-bezogenen Audit-Events gespeichert. Dies hat sich als effizienteste Implementierung erwiesen, da man in den Ressourcen-bezogenen Systemrufen die nötigen Informationen für einen direkten ACE-Zugriff bereits ermittelt hat. Aus diesem Grund und aufgrund der Tatsache das immer noch Projekte zur Implementierung einer ACL für Linux existieren, wurde bei der Implementierung der Audit-ACL von LOSA darauf geachtet, eine möglichst einfache Integrierbarkeit zu gewährleisten. Dies hat zur Folge, dass man nicht umhin kommt eine Dateisystem-ähnliche Struktur im Speicher nachzubilden.

4.1.3.1 Die Audit-ACL-Datenstruktur

Die Abbildung des Dateisystems im Speicher erfolgt nur soweit wie notwendig. Es werden nur Ressourcen gespeichert, für die eine ACL-Regel eingefügt wurde oder die sich auf dem Pfad zu einer betroffenen Ressource befinden. Die in Abbildung 10 dargestellt Struktur ist das Resultat von mindestens 7 ACL-Regeln.

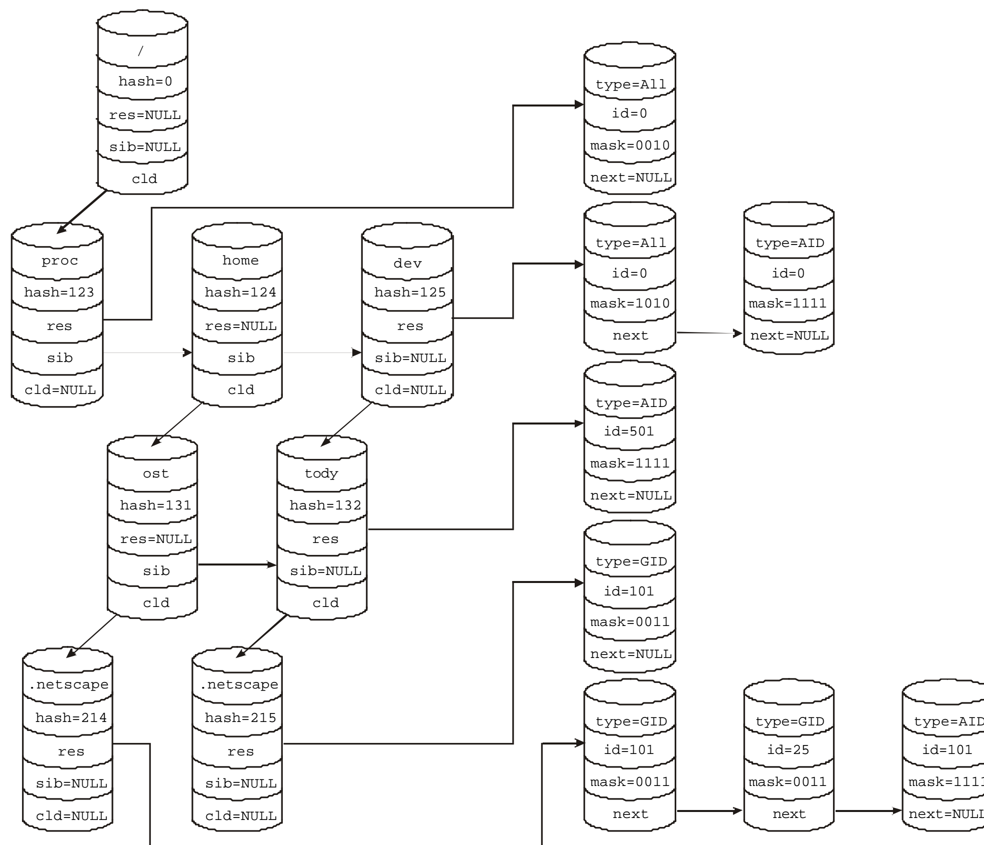

Abbildung 10 : Audit-ACL-Struktur

Der Aufbau der Audit-ACL gliedert sich in zwei Teile. Zum in die im linken Teil von Abbildung 10 dargestellten ACEs, hierbei handelt es sich um die Elemente der ACL, und die den ACEs zugeordneten RELs (*Resource Event List*) im rechter Teil von Abbildung 10. Da es nicht trivial

[10] Ob es nun daran lag, dass lange Zeit niemand die Unterstützung für ACLs in das VFS eingebracht hat oder die Verwendung von großen Dateisystemen ohne große Neuimplementierungen zwingen erforderlich wurde, sein einmal dahingestellt. Jedenfalls hat man die ACL-Einträge in den Daten-Strukturen des ext2-Dateissystems mittlerweile zur Unterstützung von Partitionsgrößen oberhalb von 2GB verwendet, so dass nicht zu erwarten ist, dass das ext2-Dateisystem jemals ACLs unterstützt.

entscheidbar ist, wann zwei ACEs eine REL gemeinsam nutzen können, besitzt jedes ACE seine eigene REL.

Audit Access Control Entry

Die ACEs werden in einer Struktur mit fünf Elementen abgelegt. Hierzu zählt der Ressourcenname und ein hash-Wert. Da Zeichenkettenvergleiche einen zu großen Zeitaufwand beanspruchen, erfolgt beim Durchsuchen der ACL zunächst ein Vergleich anhand eines hash-Wertes. Ist der hash-Wert korrekt, erfolgt zusätzlich ein String-Vergleich. Neben dem Ressourcennamen und dem hash-Wert enthält die Struktur zwei Zeiger auf ACE-Strukturen und eine Zeiger auf die REL. Wie aus Abbildung 10 ersichtlich, erfolgt über die beiden ACE-Zeiger die Nachbildung des Dateisystems, während der REL-Zeiger auf die eigentlichen Informationen des ACE verweist. Existiert keine ACE für die betroffene Ressource, aber ein ACE für eines der Verzeichnis auf dem Pfad zur Ressource, bestimmt dieses Element die Aufzeichnungsgranularität. Dabei gilt, dass in der Verzeichnisstruktur untergeordnete Einträge eine höhere Priorität besitzen.

Resource Event List

Die Struktur der REL-Elemente hat vier Einträge. Neben dem Zeiger, zur Realisierung der einfach-verketteten Liste, wird eine Event-Maske, ein Typ-Identifier und eine ID hinterlegt.

Die Event-Maske der REL-Elemente ist invers zur Event-Maske der Prozess-Struktur, d.h.:

- 00 : steht für in jedem Fall aufzeichnen,
- 01 : für das Aufzeichnen von fehlgeschlagenen Systemrufen,
- 10 : für das Aufzeichnung von erfolgreichen Systemrufen und
- 11 : für das generelle Auslassen der Aufzeichnung.

Auch wenn dies auf den ersten Blick verwirrend erscheint, so wird die Bedeutung bei der Interpretation der Audit-ACL schnell ersichtlich. Die Audit-ACL ist eine Verfeinerung der Aufzeichnungsgranularität und speichert nur Ressourcen deren Aufzeichnung verhindert werden soll. Somit besitzen alle Ressourcen, die nicht in der Audit-ACL aufgeführt sind, eine vollständig mit Nullen initialisierte Event-Maske, d.h. der Aufzeichnung wird nicht verhindert.

Da der Speicherbedarf für die ACL bei einer Vielzahl von Regeln schnell wächst, kann ein REL-Element für alle, für eine Gruppe von Nutzern, einen Nutzer oder einen bestimmten Prozess gültig sein. Hierzu existieren die Strukturelemente `type` und `id`. `type` legt die Bedeutung von ID fest. Dabei erhöht sich die Priorität mit zunehmender Verfeinerung des ID-Typs (`all` < `agid` < `aid` < `pid`), d.h. existiert für eine Ressource ein `all-` und ein `agid`-Eintrag, besitzt der `agid`-Eintrag eine höhere Priorität und bestimmt somit die Aufzeichnungsgranularität.

4.1.3.2 Einfügen von Audit-ACEs

Das Einfügen von ACEs kann mit den Funktionen `audit_init_acl(...)` und `audit_add_acl(...)` erfolgen. Beide Funktionen verwenden intern die Kernel-Funktion `audit_update_acl(...)`, sie wird im folgenden Absatz erläutert. Die Funktion `audit_init_acl(...)` bekommt eine Event-Maske übergeben und verwendet diese zum Aktualisieren eines bereits existierenden REL-Elements oder zur Initialisierung eines neuen Eintrages. Der Funktion `audit_add_acl(...)` wird eine Event-ID und der Wert für dieses Event [0..3] übergeben. Existiert bereits ein Eintrag für die betroffene Ressource, wird einzig der Wert der übergebenen Event-ID in der Event-Maske des REL-Elementes aktualisiert. Existiert noch kein Eintrag, wird eine neues Element angelegt und die Event-Maske mit Null initialisiert. Anschließend wird der Wert des übergebenen Ereignisses aktualisiert.

Die Funktion `audit_update_acl(...)` durchsucht zunächst die Audit-ACL nach der betroffenen Ressource. Fehlen Verzeichnisse auf dem Pfad zur Ressource, werden eingefügt. Da für die übergeordneten, neu eingefügten Ressourcen keine Regeln existieren, werden für diese ACEs auch

keine RELs angelegt. Ist bereits ein ACE mit einer zugehörigen REL vorhanden, wird diese REL hinsichtlich Type und ID nach einem passenden Element durchsucht. Existiert bereits ein Element, so wird dieses wie beschrieben aktualisiert. Neue Elemente werden entsprechend ihres Typs einsortiert, d.h. ein neues `agid`-Element wird hinter das letzte `agid`- und vor das erste `aid`-Element eingefügt.

4.1.3.3 Durchsuchen der Audit-ACL

Jeder überwachte Ressourcen-bezogene Systemruf ruft die Funktion `audit_check_evt(...)` zur Überprüfung der Aufzeichnungsgranularität auf. Die Funktion `audit_check_evt(...)` durchsucht die ACL entlang des Ressourcenpfades. Für jedes ACE, welches sich auf dem Pfad zur Ressource befindet, wird die zugehörige REL hinsichtlich eines passenden `pid-`, `aid-`, `agid-` oder `all`-Elementes durchsucht. Enthält die REL einen entsprechenden Eintrag, wird der zum Systemruf gehörige Wert aus der Event-Maske ausgelesen und lokal zwischengespeichert. Existiert für ein in der Verzeichnisstruktur untergeordnetes ACE ebenfalls ein passendes REL-Element, wird der zwischengespeicherte Wert aktualisiert. Existiert kein unterordnetes ACE mehr oder wurde das ACE der gesuchten Ressourcen gefunden, wird der zwischengespeicherte Wert an den Systemruf zurückgegeben. Der Rückgabewert wird mit 0 initialisiert, so dass, wenn kein Element gefunden wurde, die Aufzeichnung des Systemrufes unabhängig von dessen Ausführungsstatus durchgeführt wird.

4.1.4 Capabilities

Capabilities geben einem Objekt die Möglichkeit eine Aktion auszuführen. Im Abschnitt 3.1.3.1 wurden bereits die Capabilities des Audit-Moduls erwähnt, so hat bspw. jedes Objekt, welches die Capability `setaid` besitzt, das Privileg die Audit-ID seines Prozesses zu verändern. ohne dass die Operation für diese Struktur anschließend gesperrt wird. Bei Objekten handelt es sich immer um Prozesse. Die Identifizierung gegenüber der Capability-Datenbank erfolgt entweder über die Audit-ID des Prozesses oder über den Programmname, in dessen Kontext der Prozess ausgeführt wird. Die Capability-Datenbank ist als einfach verkettete Liste implementiert. Auf die Verwendung von speziellen Datenstrukturen zur Verbesserung der Performanz konnte hier verzichtet werden, da ein Zugriff nur bei Konfigurierungsoperationen erfolgt und deren Performanz vordergründig durch den Wechsel in der Kernel-Modus bestimmt wird.

Aufbau der Capability-List

Ein Element der Capability-Liste besteht aus vier Einträgen, der Capability abgelegt als Zahl, einer Audit-ID, einem Zeiger auf eine Programmbeschreibung und einem Zeiger auf das nächste Element der Liste. Die Programmbeschreibung wird in einer Struktur namens `audit_prog_id` abgelegt. Damit wird ein Programm durch den Programmnamen (die ersten 16 Zeichen), der Inode und der Gerätenummer, auf dem sich die ausführbare Datei befindet, eindeutig identifiziert. Die Speicherung der nötigen Informationen erfolgt im Systemruf `execve(...)`. In einem Standard-Linux-System wird nur der Programmname in der Prozess-Struktur gespeichert. Um eine eindeutige Identifizierung zu gewährleisten, wird bei LOSA zusätzlich die Inode und die Gerätenummer hinterlegt.

Ist die Capability an eine Audit-ID geknüpft, wird der Programmname in der Capability-Liste mit Nullen belegt. Anderenfalls wird die Programmbeschreibung gefüllt und die Audit-ID auf die Audit-ID des Audit-Administrators gesetzt. Dadurch wird verhindert, dass ein Capability-Eintrag sowohl für eine Audit-ID als auch für ein Programm gültig ist. (Der Audit-Administrator benötigt keine Capabilities, da dessen Audit-ID ihm per Default alle Privilegien einräumt.)

Vergabe von Capabilities

Das Hinzufügen von Capabilities erfolgt mittels der Systemrufe `audit_setcap(...)` und `audit_setcap_prog(...)`. Die Funktion `audit_setcap(...)` erwartet eine Audit-ID und eine Capability, die mittels der Defines:

- **`AU_CAP_ADM`**, erlaubt die Administration des Audit-Systems,
- **`AU_CAP_SETID`**, erlaubt das Setzen der Audit-ID und

- **AU_CAP_MSG**, erlaubt das Generieren von Audit-Ereignissen,

aus der Datei `include/linux/audit.h` spezifiziert wird. Die Funktion `audit_setcap_prog(...)` erwartet eine Capability und einen Programmnamen, wobei der Programmname mit absolutem Pfad angegeben werden muss. Die Funktion der Audit-Bibliothek ermittelt für diese Datei die Inode und die Gerätenummer und übergibt sie dem Audit-Modul. Wird das Programm anschließend verschoben, hat es all seine Capabilities verloren.

Die Vergabe von Capabilities ist im **secure**-Modus **nicht** möglich.

Abfragen der Capability

Das Abfragen der Capabilities erfolgt mittels der Funktion `audit_getcap(...)`. Die Funktion erwartet zwei Parameter, eine Audit-ID und einen Zeiger auf ein `int`-Array mit drei Elementen. Im diesem Argument werden die Capabilities der Audit-ID hinterlegt. Hierzu muss das Array mit Nullen initialisiert werden, bevor es der Funktion übergeben wird. Jedes Element steht für eine Capability (Element 0 für **admin**, Element 1 für **setaid** und Element 2 für **msg**). Enthält ein Element des Arrays nach dem erfolgreichen Aufruf von `audit_getcap(...)` einen Wert ungleich Null, so besitzt die Audit-ID diese Capability.

Zur Zeit ist es nicht möglich, die Capabilities eines Programms zu erfragen.

Löschen von Capabilities

Das Löschen der Capabilities ist zur Zeit nur durch ein Neustarten des Systems möglich. Derzeit existiert keine Funktion, die es ermöglicht einen Eintrag aus der Capability-Datenbank zu löschen.

4.1.5 Verwaltung der Audit-Daten

Bei der Verwaltung der Audit-Daten muss man zwei Arten unterscheiden. Zum einen existiert im Kernel ein Audit-Puffer, der eine bestimmte Anzahl von Audit-Records zwischenspeichert. Andererseits befinden sich die Audit-Daten in einem Audit-Trail auf der Festplatte. Die Verwaltung, der Daten im Trail ist nicht mehr Bestandteil von LOSA. Es wird zwar der im Abschnitt 3.2.6 erwähnt Trail-Viewer bereitgestellt, jedoch handelt es sich dabei nur um eine Möglichkeit die Audit-Dateien auszuwerten.

4.1.5.1 Aufbau des Audit-Puffer

Den Audit-Puffer kann man in drei Teile unterteilen. Den ersten Teil bildet eine doppelt-verkettete Liste, welche die Verwaltungsinformationen des Audit-Records und einen Zeiger auf die eigentlichen Daten enthält. Die Elemente der Liste sind vom Typ `audit_buffer`. Aus implementierungstechnischen Gründen, werden die Daten des Audit-Records in einer separaten Struktur abgelegt Hierbei handelt es sich um die Struktur `audit_record`. Den dritten Teil bildet eine einfach-verkettete Liste zur Verwaltung des Programm-Environments.

Das Programm-Environment setzt sich aus den Argumenten und der Prozessumgebung eines auszuführenden Programms zusammen, welches dem Systemruf `execve(...)` übergeben wird. Linux stellt hierzu eine maximale Größe von 128 kByte zur Verfügung. Da das `execve`-Event nur einen Teil der Audit-Events ausmacht, wäre eine statische Reservierung des Speichers uneffizient.

Abbildung 11 zeigt den Aufbau des Audit-Puffers. Die oberen drei Reihen stellen die bereits erwähnten Teile des Audit-Puffers dar. In den Arrays, im unteren Teil von Abbildung 11, ist das Programm-Environment abgelegt. Nicht jeder Programmaufruf übergibt eine Environment von 128 kByte. In den meisten Fällen reichen 4 kByte zur Speicherung der Daten aus. Um nicht unnötig Speicher zu verschwenden, wird die minimale Anzahl von Pages (je 4 kByte) anfordert, die zum Speicherung der Daten notwendig sind. Die Verknüpfung der einzelnen Pages wird im zweiten Teil des Abschnittes erläutert.

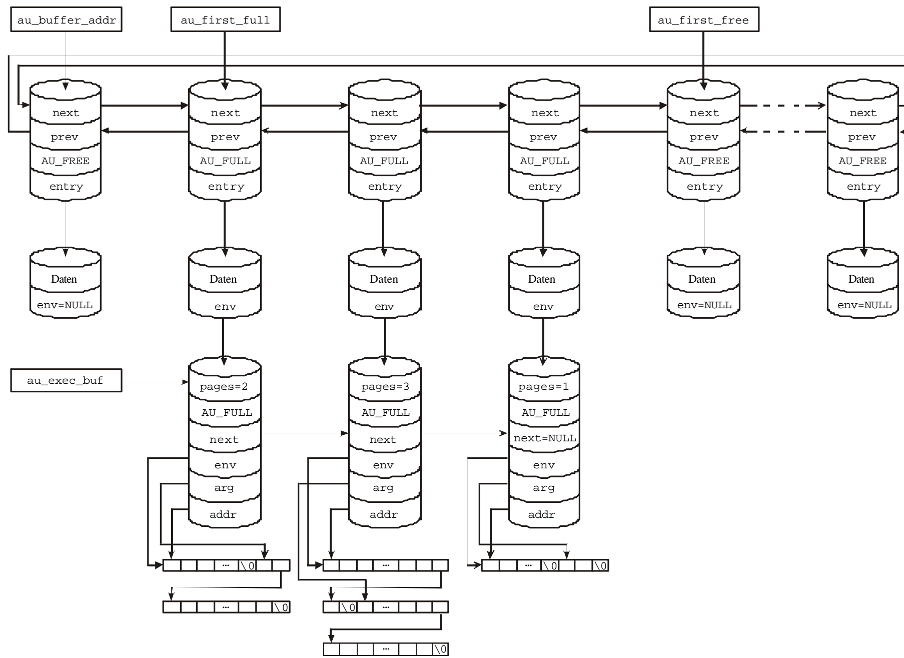

Abbildung 11 : Audit-Puffer

Zur Verwaltung der einzelnen Listen existieren die vier globalen Zeiger `au_buffer`, `au_first_free`, `au_first_full` und `au_exec_buf`.

Die Verwaltung der statischen Audit-Daten

Zum statischen Teil der Audit-Daten zählen die Strukturen `audit_buffer` und `audit_record`. Für jede `audit_buffer`-Struktur existiert eine `audit_record`-Struktur. Zusammen bilden einen Audit-Record. Bis auf das Programm-Environment besitzen alle Einträge im Audit-Record eine feste Größe. Das Programm-Environment hat eine variable Größe und wird nur aufgezeichnet, wenn der zugehörige Puffer initialisiert wurde.

Die Initialisierung des statischen Teils des Audit-Puffers erfolgt mittels der Funktion `audit_init_buffer(...)`, welche als Parameter die Anzahl der zu reservierenden Audit-Records übergeben bekommt. Der benötigte Speicher wird mit Hilfe der Kernel-Funktion `vmalloc(...)` im Gesamten angefordert. Damit wird verhindert, dass während des Aufbauens des Puffers die Prozedur abgebrochen werden muss, weil kein Speicher mehr zur Verfügung steht. Speicher der mittels `vmalloc(...)` angefordert wurde, wird nicht ausgelagert, d.h. der anforderte Speicherplatz muss physikalisch vorhanden sein. Wenn nicht genügend physikalischer Speicher zur Verfügung steht, wird die Initialisierung des Audit-Puffers abgebrochen. Nach der erfolgreichen Initialisierung des Audit-Puffers markiert der Zeiger `au_buffer` die Startadresse des angeforderten Speichers und wird im folgenden nur noch zum Löschen des Audit-Puffers verwendet.

Die Zeiger `au_first_free` und `au_first_full` dienen der Verwaltung der einzutragenden bzw. der bereits eingetragenen Audit-Events. `au_first_free` zeigt auf den ersten freien Record. Wobei ein Record frei ist, wenn der Eintrag `state` auf `AU_FREE` gesetzt ist. Zeigt `au_first_free` auf einen Record, dessen Status ungleich `AU_FREE` ist, ist der gesamte Puffer gefüllt. Sollte zu diesem Zeitpunkt eine Funktion einen neuen Record anfordern und sich das Audit-System im **`secure`**-Modus befinden, muss der anfordernde Prozess solange warten, bis mindestens ein Record freigegeben wurde. Steht ein freier Record zur Verfügung, wird das Flag `state` auf `AU_USED` gesetzt und die Adresse des Records an die anfordernde Funktion zurückgegeben. Der Zeiger `au_first_free` wird anschließend auf den Nachfolger des soeben vergebenen Records gelegt.

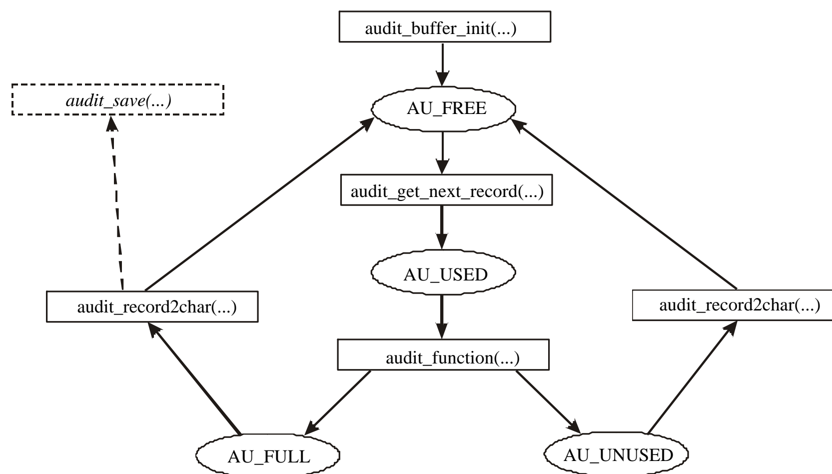

Abbildung 12 : Status eines Audit-Record

Für die Speicherung des Puffers wird der Zeiger `au_first_full` verwendet. Er "läuft" `au_first_free` hinterher. Immer wenn die Daten eines Audit-Records mittels der Funktion `audit_record2char(...)` zum Speichern in eine Zeichenkette konvertiert wurde, wird der Status der Records auf `AU_FREE` zurücksetzt. Die Funktion `audit_record2char(...)` verarbeitet nur Audit-Records, deren Status `AU_FULL` oder `AU_UNUSED` ist. Ein Audit-Record wird als `AU_FULL` gekennzeichnet, wenn der Audit-Funktion des Systemrufes alle Daten in die Struktur eingetragen wurden. Als `AU_UNUSED` wird ein Record gekennzeichnet, wenn er mittels der Funktion `audit_get_next_record(...)` angefordert wurde, aber keine Daten eingetragen wurden. In diesem Fall wird der Audit-Record wiederfreigegeben, aber nicht abgespeichert, da die Struktur ungültige Daten enthält.

Die Verwaltung des Programm-Environments

Für eine spätere Analyse sind Informationen über das Programm-Environment eines Prozesses mitunter besonders hilfreich. Es hält Informationen, wie und in welchem Kontext ein Programm ausgeführt wurde. Die Größe des Programm-Environments kann bis 128 kByte betragen, so dass eine statische Reservierung des benötigten Speichers unmöglich wird. Die Audit-Modul stellt hierfür eine eigene Pufferverwaltung zur Verfügung, die mit dem Audit-Puffer eng verbunden ist.

Der Environment-Puffer besteht aus einer einfach verketteten Liste von `audit_exec_buffer`-Strukturen. Die Struktur enthält 6 Einträge:

- `pages`, enthält die Anzahl der bereits allokierten Speicherseiten,
- `state`, beschreibt den aktuellen Zustand des Records,
- `next`, zeigt auf den nächsten Record,
- `env`, zeigt auf den Beginn der Environment-Zeichenkette,
- `arg`, zeigt auf den Beginn der Argumenteszeichenkette und
- `addr`, verweist auf die Startadresse der ersten Speicherseite des Audit-Records.

Zur Ablage der Zeichenketten werden aus Performanzgründen ganze Speicherseiten allokiert. Die Größe einer Speicherseite beträgt unter Linux mindestens 4 kByte[11]. Mittels des Parameters `order`, der bei der Initialisierung des Environment-Puffers von Administrator angegeben und beim Allokieren der Seiten der Funktion `get_free_pages(...)` übergeben muss, wird die Anzahl der

[11] Die gilt nur für eine 32 Bit Architekturen. Auf einem 64 Bit Rechner (Digital Alpha, UltraSparc, ...) beträgt die Größe einer Speicherseite mindestens 8 kByte.

bereitzustellenden Speicherseiten angegeben. Der Wert (`order`) kann zwischen 0 und 5 liegen, woraus sich die Größe des angeforderten Speichers wie folgt berechnet:

```
buffer_size = page_size * 2 ^ order.
```

Der festgelegte Wert für `order` gilt global für alle Records des Environment-Puffers.

Das Element von `pages` enthält die Anzahl der Speicherseiten des Puffers, die sich hinter der Adresse `addr` verbergen. Im Normalfall, sollte der Wert von `pages` gleich Eins sein. Sollte das Environment nicht in die bereitgestellten Speicherseiten passen, muss erneut Speicher angefordert werden. Die Verknüpfung der beiden Seiten erfolgt über die letzten vier Bytes der vorhandenen Seite. Hier wird die Adresse der neuen Seite eingetragen, deren letzte 4 Bytes mit Nullen initialisiert werden. Sie markieren das Ende des gesamten Puffers. Sollte der Speicherplatz wiederum nicht ausreichen, wird noch eine weitere Speicherseite angefordert. Dies geschieht solange, bis 128 kByte angefordert wurden.

Da die Verknüpfung der einzelnen Seiten aufwendig zu realisieren ist, so dass sowohl beim Eintragen der Daten als auch beim Speichern für jeden zusätzlich angeforderten Speicherbereich ein gewisser Overhead entsteht, obliegt es dem Systemadministrator, einen richtigen Wert für `order` zufinden. Es ist jedoch unbedingt zu beachten, dass der Speicher für den Environment-Puffer nicht ausgelagert wird und somit physikalisch vorhanden sein muss. Ist die Funktion `get_free_pages(...)` nicht in der Lage die nötige Anzahl von aufeinanderfolgenden freien Speicherseiten bereitzustellen, wird das Programm-Environment nicht aufgezeichnet. Mitunter ist es sinnvoller einen kleineren Wert für `order` zu wählen und den Overhead in Kauf zu nehmen, bevor man wichtige Informationen verliert.

Bei der Initialisierung des Environment-Puffers muss der Administrator außerdem die Anzahl der zu puffernden Environment-Records angeben. Beim Speichern der Audit-Records wird überprüft, ob die aktuelle Anzahl an Records größer als der vorgegebene Wert ist. Ist dies der Fall, werden nach dem Speichern so viele freie Records aus der Liste entfernt, bis dessen Größe dem vorgegebenen Wert entspricht. Hierbei ist, wie bei der Angabe der Page-Order, ein richtiger Wert durch testen zu ermitteln. Die Anzahl der Records sollte nicht zu klein gewählt werden, da das Erzeugen einen nicht unbedeutenden Aufwand erfordert. Allerdings kann bei einem großen Environment-Puffer, der Speicherbedarf für diesen schnell anwachsen und somit den User-Speicher stark vermindern.

4.1.5.2 Generierung eines Audit-Ereignissen

Das Generieren von Audit-Daten kann auf zwei Arten initiiert werden. Durch den Aufruf eines der überwachten Systemruf oder durch die Verwendung des Systenrufes `sys_audit_event(...)`. Mittels dies Systemrufes können Anwendungsprogramm die Generierung eines Audit-Events initiieren.

Das Erzeugen eines Audit-Events beginnt immer im Systemruf. Zunächst wird mittels der Funktion `audit_doit(...)` anhand der Präselektionsmaske des Prozesses überprüft, ob die Protokollierung des Ereignisses erfolgen soll. Außerdem überprüft die Funktion `audit_doit(...)`, ob das Betriebssystem-Audit aktiviert ist und ein Audit-Dämon läuft. Ist dies nicht der Fall, wird die Erzeugung des Events an dieser Stelle abgebrochen. Die Überprüfung des Status des Audit-Moduls und des Audit-Dämons erfolgt so zeitig wie möglich, um bei deaktiviertem Audit so wenig wie möglich Overhead zu erzeugen. Erst wenn sichergestellt ist, dass die Protokollierung aktiviert ist, wird mittels der in der Task-Struktur enthalten Preselektionsmaske überprüft, ob für diesen Nutzer, für dieses Ereignis und für diesen Ausführungsstatus des Systemrufs ein Audit-Event generiert werden soll.

Wurde sichergestellt, dass ein Audit-Event für diesen Nutzer, für dieses Ereignis und für diesen Ausführungsstatus erzeugt werden soll, wird die zum Systemruf gehörige Audit-Funktion aufgerufen. Es existieren zwei Arten von Audit-Funktionen. Zum einen Funktionen, die einem Systemruf, der sich auf eine Ressource des Dateisystems beziehen (bspw. `open(...)`), zugeordnet sind und Funktionen,

die Systemrufe, die die Eigenschaften des Systems oder eines Prozesses verändern (bspw. `fork(...)`, `setuid(...)`), verarbeiten. Bei Funktionen, die sich auf eine Ressource beziehen, muss zunächst die Ressource mittels der Funktion `audit_resolve_filename(...)` genau bestimmt werden. Hierzu muss der Ressourcenname in den Kernel-Speicher kopiert werden. Um ein unnötiges Allokieren von Speicher zu verhindern, wird zunächst mittels der Funktion `audit_get_free_record(...)` ein freier Audit-Record angefordert und der Ressourcenname direkt in diesen Audit-Record kopiert.

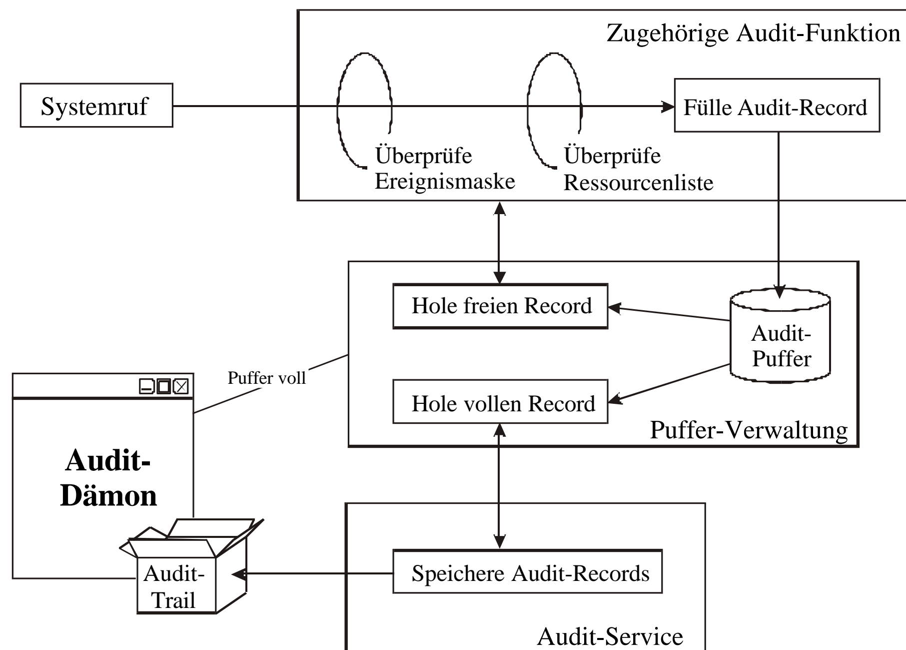

Abbildung 13: Generierung eines Audit-Records

Wurde der Dateiname in den Audit-Record und damit in den Kernel-Speicher kopiert und der Pfad mittels der Funktion `audit_resolve_filename(...)` vollständig aufgelöst, kann mit Hilfe der ACL die Auswahl anhand des Ressourcennames, des Nutzers, des Ereignisses und des finalen Ausführungsstatus erfolgen. Hierzu muss in der ACL ein Eintrag für die betroffene Ressource existieren. Ist kein Eintrag vorhanden, wird der Audit-Record mit den verbleibenden Informationen gefüllt und die Protokollierung erfolgreich abgeschlossen. Enthält die ACL einen Eintrag für diese Ressource, diesen Nutzer und dieses Ereignis, wird innerhalb der Audit-Funktion der finale Ausführungsstatus überprüft. Stimmt dieser ebenfalls überein, wird die Generierung des Audit-Events an dieser Stelle abgebrochen. Hierzu wird der Status des Audit-Records, damit mit er von der Funktion `audit_record2char(...)` überlesen wird, auf `AU_UNUSED` gesetzt.

Zum Eintragen der Informationen in den Audit-Record stehen drei Funktionen zur Verfügung:

- **`audit_fill_static(...)`**, fügt die für alle Audit-Events geltenden Einträge ein, hierbei handelt es sich um `date`, `hostname`, `aid`, `ruid`, `euid`, `rgid`, `egid`, `pid`, `tty`, `sid`, `from`.
- **`audit_fill_cfilename(...)`**, füllt alle Einträge, die sich auf die Ressource (`file`) beziehen, dabei handelt es sich um `inode`, `dev`, `owner`, `gowner`, `perm`, die Ressource muss als Zeichenkette übergeben werden und
- **`audit_fill_filename(...)`**, analog `audit_fill_cfilename(...)`, allerdings wird die Ressource direkt mittels der Inode übergeben.

Alle Daten, die mit diesen Funktionen nicht eingetragen werden können, werden durch die Audit-Funktion direkt in den Record eingefügt. Abschließend wird der Status des Audit-Records auf

AU_FULL gesetzt, auf diese Weise wird der Puffer-Verwaltung signalisiert, dass dieser Record gespeichert werden kann.

Dieser Ablauf gilt bis auf wenige Ausnahmen für alle überwachten Systemrufe. Ausgenommen sind Systemrufe, die während ihrer Ausführung Audit-relevante Informationen löschen oder verändern. Um dies zu verhindern, existieren für diese Systemrufe jeweils zwei Audit-Funktionen. Die erste Audit-Funktion wird zu Beginn des Systemrufes aufgerufen. Sie fordert einen freien Audit-Record an und füllt diesen mit allen bereits vorhandenen Informationen. Die zweite Audit-Funktion wird nach der vollständigen Abarbeitung des Systemrufes aufgerufen und fügt den finalen Ausführungsstatus und alle verbleibenden Informationen ein. Soll das Ereignis für diesen finalen Ausführungsstatus nicht protokolliert werden, wird lediglich der Status des Audit-Records auf AU_UNUSED gesetzt.

4.1.5.3 Speicherung der Audit-Daten

Das Speichern der Audit-Daten wird von Audit-Dämon initiiert. Das Audit-Modul sammelt die Audit-Events bis zu einem bestimmten Schwellwert (watermark). Der Schwellwert ist in der globalen atomaren Variable[12] au_watermark gespeichert. In der Regel liegt der Wert bei 90 Prozent der Audit-Records im Puffer. Wird der Puffe bis zu diesem Wert gefüllt, wird ein Speichern der Audit-Records notwendig. Da der Kernel nicht selbst entscheiden kann, wo die Daten abgelegt werden sollen, wird von der Funktion audit_get_free_record(...) das Signal SIGALRM an den Audit-Dämon gesendet. Dieser ruft daraufhin die Funktion audit_service(...) mit den Parametern:

- fd, der Dateideskriptor der aktuellen Audit-Datei und
- limit, dient der Einhaltung der Trail-Limits,

auf. Das Speichern der Audit-Records erfolgt wiederum komplett im Kernel. Dies hat den Vorteil, dass sich die Audit-Daten zu keiner Zeit im User-Speicher befinden. Der Parameter limit wird vom Audit-Dämon vor jedem Aufruf von audit_service(...) berechnet, hierzu wird die Größe der aktuellen Audit-Datei und der auf der Partition verfügbare Speicherplatz ermittelt. Das daraus errechnet Limit wird der Funktion audit_service(...) übergeben. Wird während des Speichers der Records das Limit überschritten, bricht die Funktion das Speichern ab und liefert den Fehler E2BIG zurück. Der Audit-Dämon versucht daraufhin, wie im Abschnitt 3.2.4.3 beschrieben, eine neue Audit-Datei zu öffnen und die verbleibenden Records zuspeichern.

In der Funktion audit_service(...) wird jeder Record einzeln verarbeitet und in die Audit-Datei gespeichert. Um beim Speichern der Audit-Daten unabhängig von Datensystem zu bleiben und nicht zuviel Performanz zu verlieren, wird eine Kernel-interne-write-Funktion verwendet. Mittels des Dateideskriptors lässt sich die Inode und damit die Inode-Struktur des Dateisystems ermitteln. In der Inode-Struktur existiert ein Zeiger auf die zugehörige write-Funktion, welche zum Speichern der Audit-Daten verwendet wird. Das Auflösen der richtigen Funktion erfolgt einmal zu Beginn der Funktion audit_service(...) und ist äquivalent dem Vorgehen der write-Funktion des VFS.

Allerdings besteht bei der Verwendung der write-Funktion das Problem, dass sich die zu speichernden Daten im Nutzerspeicher befinden müssen. Aus diesem Grund muss beim Initialisieren des Audit-Moduls ein entsprechender Bereich vom Audit-Dämon bereitgestellt. Es ist in jedem Fall darauf zu beachten, dass der bereitgestellte Puffer groß genug ist. Im Kernel erfolgt keine Überprüfung des Speicherbereiches. Ist der Puffer zu klein gewählt, wird das gesamte System instabil und kann unter Umständen abstürzen.

[12] Im Linux-Kernel existiert hierfür der Datentyp atomic_t. Unter der Verwendung der dazugehörigen Funktionen ist auch in einem SMP-System sichergestellt, dass der Wert einer solchen Variable zu jedem Zeitpunkt im gesamten System gleich ist.

4.2 Aufbau des Audit-Trails

Die Speicherung der Audit-Daten erfolgt binäre, so dass es nicht möglich ist, den Audit-Trail ohne ein spezielles Programm auszulesen. Im Abschnitt 3.2.6 wurde bereits der Trail-Viewer erwähnt, jedoch ist dies nur eine Möglichkeit die Audit-Daten auszuwerten. Um dem Anwender die Möglichkeit zu geben, den Audit-Trail in eigenen Anwendungen auszulesen, wird der Aufbau des Trail in diesem Abschnitt etwas ausführlicher erläutert.

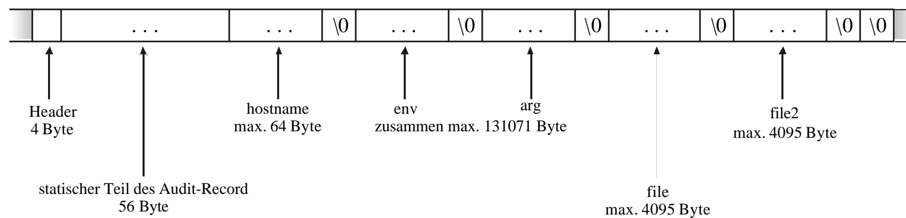

Abbildung 14 : Aufbau des Audit-Trails

Der Audit-Trail setzt sich aus einzelnen Audit-Records, die hintereinander angeordnet sind, zusammen. Abbildung 14 zeigt das Format eines Records, der im folgenden erläutert wird.

header

Der Header kennzeichnet den Beginn eines Audit-Records und ist ein vier Byte long-Wert. Im Moment werden einzig Records von Typ *`host`* abgespeichert. Die darauffolgenden Daten beinhalten die Informationen zum Record. Das Format ergibt sich implizit aus dem Record-Typ. Einzige die abschließende Stringendkennung ist bei allen Formaten gleich. Der Header und die Stringendkennung schließen einen Audit-Record ein.

statischer Teil des Audit-Records

Der statische Teil des Audit-Records ist eine Kopie der ersten 56 Bytes des Audit-Records. Es wird keine Typenkonvertierung vorgenommen, so dass ein `short`-Wert auch nur zwei Byte in der Audit-Datei belegen. Solange das Audit-System auf einer x86-Architektur eingesetzt wird, ist dies unproblematisch. Bei einer Portierung auf eine andere Architektur muss an dieser Stelle unter anderem die Ausrichtung der Datentypen beachtet werden.

`hostname`

Der Rechnername wird als Zeichenkette abgelegt und ist auf Länge von 64 Zeichen beschränkt. Das Define `MAXHOSTNAMELEN` gibt zwar einen Wert von 65 Byte vor, jedoch ist hier nicht die abschließende Stringendkennung mit eingerechnet. Der Rechnername sowie alle anderen Zeichenketten dürfen keine Stringendkennungen enthalten, da das Format des Records durch diese beschrieben wird und deren Anzahl damit relevant ist.

`env` und `arg`

Das Programm-Environment wird innerhalb Audit-Moduls bereits in eine Zeichenkette konvertiert. Die einzelnen Argumente sind durch ein Leerzeichen getrennt. Die Länge von `env` und von `arg` ist nicht näher bestimmt. Einzig beide Einträge zusammen dürfen nicht mehr als 128 kByte belegen. Beim Auslesen der Audit-Datei erweist es sich als sinnvoll, gleich einen Puffer von 128 kByte anzulegen und dort beide Zeichenketten hineinzukopieren.

`file` und `file2`

`file` und `file2` sind ebenfalls Zeichenketten und enthalten die Dateinamen bei Ressourcen-bezogenen Audit-Events.

Die Angabe von `env`, `arg`, `file` und `file2` ist nicht bei jedem Audit-Event geben. Sind keine Informationen vorhanden, bleiben die Zeichenketten leer und nur die Stringendkennungen werden zur Beschreibung des Record in der Audit-Datei abgelegt.

4.2.2 Ausnahmen bei connect- bzw. accept-Ereignissen

Die Audit-Events `AUE_CONNECT` und `AUE_ACCEPT` benötigen zur Speicherung der Daten zwei Zahlenwerte und ein IP-Adresse. Da die IP-Adresse allerdings abhängig vom verwendeten Protokoll ist, musste an dieser Stelle etwas getrickst werden. Das Argument 1 beschreibt das verwendete Protokoll, hierbei sind die Protokolle `AF_UNIX` (UNIX-Domain-Sockets), `AF_INET` (Internet-Socket im IPv4-Format) und `AF_INET6` (Internet-Socket im IPv6-Format) möglich. UNIX-Domain-Sockets kommunizieren über eine benannte Pipe, welche im Record wie eine normale Datei betrachtet und abgespeichert werden. Internet-Sockets werden durch die IP-Adresse und den Port beschrieben. Der Port wird im Argument 2 hinterlegt und die IP-Adresse binäre im Record-Eintrag `file` abgespeichert.

Die binäre Speicherung der IP-Adresse bürgt jedoch das Problem, dass unter Umständen mehrere Stringendkennungen (binäre Speicherung der Zahl Null) im Eintrag `file` abgelegt werden. Da das Einlesen des Records anhand der Stringendkennungen synchronisiert wird, muss nun jeder Record hinsichtlich des Event-Eintrages überprüft werden. Beinhaltet der aktuelle Record ein Event mit der ID `AUE_CONNECT` oder `AUE_ACCEPT`, muss das Argument 1 hinsichtlich des Protokolls überprüft werden. Davon abhängig wird der Eintrag `file` eingelesen:

- `AF_UNIX`: der Eintrag wird als normale Datei betrachtet,
- `AF_INET`: es werden genau 4 Byte eingelesen bzw.
- `AF_INET6`: es werden genau 16 Byte eingelesen.

Dies ist zwar keine besonders schöne Implementierung, sie hat jedoch den Vorteil, dass der Performanzverlust durch die anfallenden Vergleichsoperationen auf der Seite des auslesenden Prozesses liegt. Im Audit-Modul können die Werte ohne tiefergehende Unterscheidungen in den Audit-Trail geschrieben werden.

4.3 Kommunikation zwischen Audit-Dämon und Steuerungsprogramm

Wie bereits einleitend erwähnt, ist der Audit-Dämon für die Verwaltung der Audit-Trails und die Administrierung der Audit-Puffer im Audit-Modul verantwortlich. Da dieser jedoch kein Ein- und kein Ausgabeterminal besitzt, muss eine interaktive Bedienung des Audit-Dämon mittels des Steuerungsprogramms erfolgen. Hierbei werden Signale und benannte Pipes zur Kommunikation verwendet.

Der Audit-Dämon reagiert auf folgende Signale:

- **`SIGTERM`**, beendet den Audit-Dämon.
- **`SIGHUP`**, veranlasst den Wechsel der Audit-Datei.
- **`SIGALRM`**, wird von Kernel gesendet und signalisiert dem Audit-Dämon, die Funktion `audit_service(...)` aufzurufen.
- **`SIGUSR1`**, veranlasst das erneute Einlesen der Konfigurationsdatei.
- **`SIGUSR2`**, dient der Synchronisation der Kommunikation der interaktiven Shell.

Die für das Versendung von Signalen benötigte Prozess-ID (`pid`) wird vom Audit-Dämon laut der Konfigurationsregel `connect` in einer Datei gegelegt. Normalerweise wird hierfür die Datei `/var/run/auditd.pid` verwendet. Neben der Prozess-ID legt der Audit-Dämon in dieser Datei auch den Dateinamen der ersten interaktiven Pipe mit absolutem Pfad ab. Dabei ist der Name und die Position der Datei durch die Konfigurationsregel `pipe` festgelegt.

Alle neuen Einträge in der `connect`-Datei werden ans Ende angehangen. Wird ein zweiter Audit-Dämon gestartet, trägt dieser sein Prozess-ID hinter die `pid` des ersten Dämons ein. Die Prozess-ID des laufenden Audit-Dämons bleibt erhalten und kann vom Steuerungsprogramm korrekt ausgelesen werden. Die Datei verbleibt solange auf dem Datenträger, bis der Audit-Dämon beendet wird, der seine `pid` als erstes eingetragen hat.

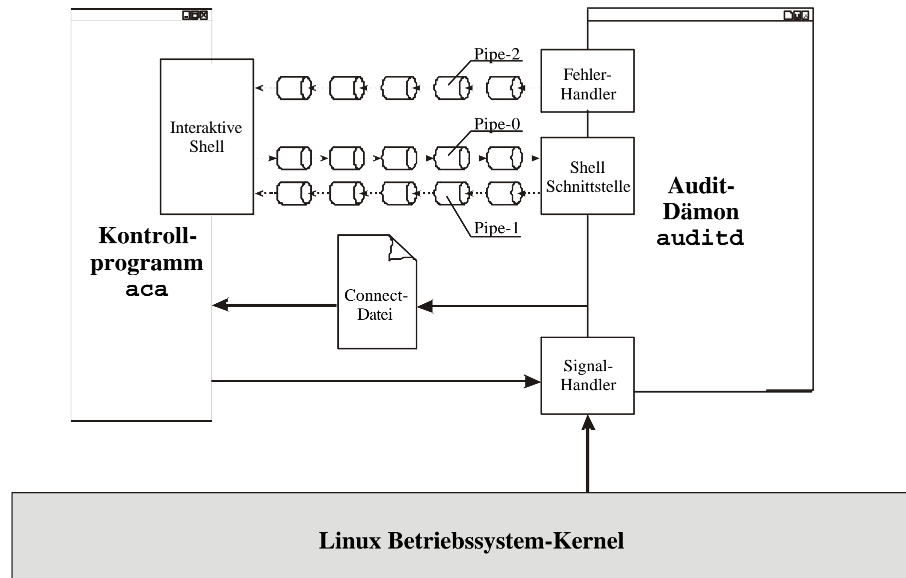

Abbildung 15 : Kommunikation zwischen Audit-Dämon und Steuerungsprogramm

Abbildung 15 zeigt die einzelnen Kommunikationskanäle zwischen dem Audit-Dämon und dem Steuerungsprogramm. Eine Pipe ist unidirektional, so dass für die interaktive Shell des Steuerungsprogramms mindestens zwei Pipes erforderlich sind. Die dritte Pipe wird zur Ausgabe von Fehlermeldungen verwendet, so diese an die Shell geschickt werden sollen.

Verbindungsaufbau zur interaktiven Shell

Der Aufbau der Verbindung über die benannten Pipes muss nach einem festen Schema erfolgen. Da der Audit-Dämon nicht in der Lage ist, auf irgendetwas außer auf ein Signal zu warten, wird der Aufbau über das Signal **`SIGUSR2`** gesteuert. Der hier beschriebene Ablauf wird vom Steuerungsprogramm (`aca`) ausgeführt. Durch das Modul-Konzept ist der Audit-Dämon allerdings nicht auf dieses Programm beschränkt. Eine Applikation, welche die Verbindung wie folgt beschrieben aufbaut, ist ebenfalls in der Lage die Steuerung des Audit-Dämon zu übernehmen.

Der Aufbau der Verbindung gliedert sich in drei Schritte. Zunächst müssen die beiden Pipes mittels des Systemrufs `mkfifo(...)` erstellt werden. Dem Audit-Dämon ist nur der Name der Pipe-0 bekannt, welche er zum Lesen verwendet. Der Pfadname der Pipe-1 wird im Laufe des Verbindungsaufbaus an den Audit-Dämon übermittelt. Um sicherzustellen, dass nicht jeder mit dem Audit-Dämon Verbindung aufnehmen kann, muss der Besitzer der ersten Pipe `root` oder mit dem Nutzer, der den Audit-Dämon gestartet hat, identisch sein.

Konnten beide Pipes erfolgreich erstellt werden, wird der Audit-Dämon mittels des Signals **`SIGUSR2`** darüber informiert. Dieser öffnet anschließend die Pipe-0 zum Lesen. Das Steuerungsprogramm wartet während dessen mittels des Systemrufs `open(...)` auf den Audit-Dämon, bis dieser die Pipe-0 erfolgreich geöffnet hat. Damit ist der Schritt des Verbindungsaufbaus abgeschlossen.

Der Anschluss an die zweite Pipe wird wiederum von Steuerungsprogramm initiiert. Zunächst öffnet dieses die Pipe-1 zum Lesen ohne zu blockieren und schreibt in die Pipe-0 die genaue Position der Pipe-1. Anschließend wird mittels des Signals **`SIGUSR2`** der Audit-Dämon darüber informiert, dass

dieser den Verbindungsaufbau fortsetzen kann. Nach dem Absenden des Signals wartet das Steuerungsprogramm mittels des Systemrufes `read(...)` auf den Audit-Dämon. Dieser liest den Namen der Pipe-1 aus der Pipe-0 und öffnet die Pipe-1 zum Schreiben. Um dem Steuerungsprogramm den erfolgreichen Aufbau mitzuteilen, wird die Zeichenkette `okay` in die Pipe-1 geschrieben. Nachdem dieses die Nachricht empfangen hat, ist der Verbindungsaufbau erfolgreich abgeschlossen.

Um mit dem Audit-Dämon über diese Verbindung zu kommunizieren, schreibt das Steuerungsprogramm das Kommando in die Pipe-0 und sendet das Signal `SIGUSR2` an den Audit-Dämon, dieser bestätigt jedes Kommando mit der Zeichenkette `okay`.

Verwendung der Error-Pipe (Pipe-2)

Zunächst muss die Pipe-2 vom Steuerungsprogramm mittels `mkfifo` angelegt werden. Anschließend wird das Kommando:

```
getout [pathname_of_pipe-2]
```

mittels der Pipe-0 an den Audit-Dämon übertragen. Der Audit-Dämon versucht anschließend die Error-Pipe zu öffnen. Solange wie eine Verbindung über die Error-Pipe zwischen dem Audit-Dämon und dem Steuerungsprogramm existiert, schreibt der Audit-Dämon alle Meldungen, die normalerweise an den Syslog-Dämon gesendet werden, zusätzlich in die Pipe-2.

Das Auslesen muss vom Steuerungsprogramm organisiert werden. In der Implementierung des `aca` erfolgt dies mittels einer `select`–Anweisung über `stdin` und über die Pipe-2. Tritt beim Übertragen einer Meldung über die Error-Pipe ein Fehler auf, schließt der Audit-Dämon die Pipe. Sollte das Steuerungsprogramm die Informationen nicht rechtzeitig auslesen, kann es passieren, dass die Pipe voll ist und der Audit-Dämon diese schließt.

Die Error-Pipe kann nur verwendet werden, wenn eine Kommunikation über die interaktive Shell erfolgt. Dies sichert, dass die Informationen lediglich an ein bereits autorisiertes Programm gesendet werden können.

4.4 Betriebsmodus

Die Verwaltung des Betriebsmodus ist Teil des Sicherheitskonzeptes von LOSA und erfolgt zusammen mit der Administration der Capabilities innerhalb der Funktion `audit_cap_ctl(...)`. Der aktuelle Modus wird im Audit-Modul in der globalen Variable `au_modus` gespeichert und kann mit folgenden Defines belegt werden:

- `AU_DISABLED`: das Audit-System ist deaktiviert,
- `AU_NORMAL`: Konfigurationsmodus
- `AU_SECURE`: normaler Betriebsmodus

Das Setzen des Betriebsmodus kann mittels des Kernel-Parameters **`audit`** zur Bootzeit oder mittels der Funktion `audit_chmod(...)` zur Laufzeit erfolgen.

Setzen des Betriebsmodus zur Laufzeit

Das Setzen des Betriebsmodus kann zur Laufzeit mittels der Funktion `audit_chmod(...)` erfolgen. Die Funktion erwartet ein Argument, dass einem der oben beschriebenen Defines entsprechen muss. Befindet sich das Audit-System im **`secure`**-Modus, ist das Verändern des Betriebsmodus nicht mehr möglich. Der **`secure`**-Modus kann nur einen Neustart des Rechners verlassen werden.

Kernel-Parameter

Außer mit der Funktion `audit_chmod(...)` kann der Betriebsmodus des Audit-Systems mittels des Kernel-Parameters **`audit`** gesetzt werden. Bei der Verwendung des Parameters ist folgende Syntax einzuhalten:

```
audit=[modus],[audit-admin]
```

`modus`: gibt den Betriebsmodus an und muss einer der Zeichenketten **`disabled`**, **`normal`** oder **`secure`** entsprechen. Sollte das System im **`secure`**-Mode oder im Modus **`disabled`** gestartet werden, ist es nicht möglich das Audit-System zu konfigurieren.

`audit-admin`: ist die Audit-ID des Audit-Administrators, der Parameter ist optional. Ist keine Audit-ID angegeben, wird automatisch die Nutzer-ID 0 verwendet. Zur Laufzeit ist es nicht möglich die Audit-ID des Audit-Administrator zu verändern.

Abfragen des aktuellen Betriebsmodus und des Audit-Administrators

Zum Erfragen des Betriebsmodus wird die Funktion `audit_getmod(...)` zur Verfügung gestellt. Die Funktion erwartet kein Argument und liefert den aktuellen Betriebsmodus des Audit-Systems zurück. Befindet sich das Audit-System im **`secure`**-Modus benötigt der aufrufende Prozess die Capability **`admin`**.

Das Abfragen des aktuellen Audit-Administrators erfolgt mittels der Funktion `audit_getadmin(...)`. Die Funktion erwartet die Adresse einer `int`-Variable, in welcher die Audit-ID des aktuellen Audit-Administrators abgelegt wird. Die Audit-ID kann nicht mittels des Rückgabewertes zurückgegeben werden, da die Audit-ID positive als auch negative Werte annehmen kann und somit die Unterscheidung zwischen einem erfolgreichen und einem fehlgeschlagen Aufruf nicht möglich ist. Befindet sich das Audit-System in **`secure`**-Modus benötigt der aufrufende Prozess die Capability **`admin`**.

4.5 Start-Skript `audit`

Wie jeder Dienst in einem UNIX-System wird auch der Audit-Dämon über eine sogenanntes init-Skript gestartet. Bei einem RedHat-System, sowie bei den meisten anderen Linux-Distributionen, befinden sich die init-Skripte im Verzeichnis `/etc/rc.d/init.d/`. Im Verzeichnis des jeweiligen Runlevels existieren ein oder zwei Links auf diese Skripte. Links mit einem Namen wie `Sxx[name]`, wobei `xx` für eine Zahl und `name` für den eigentlichen Name des Skriptes steht, sind Startskripte. Links mit einem Name wie `Kxx[name]`, sind Kill-Skripte, sie werden beim Herunterfahren des Systems ausgeführt. Je niedriger die Zahl ist, je eher werden die Skripte beim Hoch- bzw. Herunterfahren des Systems ausgeführt. Das Audit-Skript sollte so früh wie möglich ausgeführt werden. Die zugehörigen Links erhalten aus diesem Grund immer den Name `S01audit`. Das Anlegen eines Kill-Skriptes ist nicht notwendig, da der Audit-Dämon ordnungsgemäß durch das Kill-Signal des Systems beendet wird.

Das Audit-Skripte erwartet einen Parameter, der die auszuführende Aktion spezifiziert. Folgende Parameter sind möglich:

- `start`: der Audit-Dämon wird gestartet,
- `stop`: der laufende Audit-Dämon wird beendet oder
- `restart`, `reload`: eine laufender Audit-Dämon wird beendet und anschließend neu gestartet.

Beim Aufruf von `audit stop` wird dem Audit-Dämon lediglich durch ein Kill-Signal gesendet. Aus diesem Grund ist es auch nicht notwendig, extra eine kill-link anzulegen, da das System beim

Herunterfahren an alle Prozesse ein Kill-Signal sendet. Das Beenden des Audit-Dämon ohne die Verwendung des Skriptes hat den Vorteil, dass das Protokollierung zum spätmöglichsten Zeitpunkt deaktiviert wird.

Der Aufruf von `restart` bzw. `reload` ist nichts anderes als `audit stop` und `audit start` in einem Schritt.

Der Aufruf von `audit start` ist wesentlich umfangreicher und wird im folgenden erläutert.

4.5.1 Init `audit start`

Wird das Skript `audit` mit dem Argument `start` aufgerufen, erfolgt die Initialisierung des Audit-Systems entsprechend der Konfigurationsdatei `/etc/security/audit.conf`. Die Initialisierung lässt sich in 8 Schritte unterteilen:

Schritt 1: Überprüfen der Voraussetzungen

Damit das Audit-System korrekt gestartet werden kann, müssen folgende Bedingungen erfüllt sein:

- Das Steuerungsprogramm muss unter `/usr/sbin/aca` installiert und ausführbar sein.
- Das Audit-System darf sich nicht im Modus **`disabled`** befinden.
- Die Konfigurationsdatei muss sich unter `/etc/security/audit.conf` befinden und lesbar sein.

Ist eine der Bedingungen nicht erfüllt, wird die Initialisierung des Audit-Systems an dieser Stelle abgebrochen. Sollte das Audit-System abweichend installiert werden, muss das init-Skript entsprechend angepasst werden.

Schritt 2: Einlesen der Konfigurationsdatei

Die Konfigurationsdatei wird zeilenweise eingelesen. Für jede mögliche Regel existiert eine Variable im Skript. Wurde eine Regel erkannt, wird die Variable entsprechend gesetzt. Alle Variablen werden mit einer leeren Zeichenkette initialisiert. Wurde eine Variable beim Parsen der Konfigurationsdatei nicht gesetzt, wird die entsprechende Aktion nicht ausgeführt.

Beim Einlesen der Konfigurationsdatei werden keine Syntaxfehler beachtet. Ist die Konfigurationsdatei syntaktisch inkorrekt, ist der Zustand, in dem sich das Audit-System nach der Initialisierung befindet, unbestimmt. Aus diesem Grund sollte jede neue Konfiguration zuvor getestet werden, bevor das System produktiv eingesetzt wird. Dies ist besonders wichtig, wenn das Audit-System nach dem Ablauf der Initialisierung in den **`secure`**-Mode wechselt.

Schritt 3: Überprüfen des Parameters `boot`

Ist der Parameter `boot` auf den Wert `disabled` gesetzt, wird die Initialisierung des Audit-Systems in diesem Schritt abgebrochen. Alle anderen Parameter der Konfigurationsdatei bleiben unbeachtet. Dies gilt auch fir weitere `boot`-Regel, die in einer vorherigen Zeile der Konfigurationsdatei enthalten ist. Dadurch wird eine einfache Möglichkeit geschaffen, das Audit-System zu deaktivieren. Man muss lediglich am Ende der Konfigurationsdatei die Zeile `boot = disabled` einfügen. Soll das System wieder aktiviert werden, muss nur diese Zeile auskommentiert werden.

Schritt 4: Initialisierung aller bereits laufenden Prozesse

Bis zu diesem Zeitpunkt sind bis auf den Betriebsmodus und die Audit-ID des Audit-Administrators keine Variablen des Audit-Systems initialisiert. Beide besitzen einen default-Wert oder wurden mittels eines Kernel-Parameters gestetzt. Allerdings laufen zu diesem Zeitpunkt schon einige Prozesse, die später in die Überwachung mit einbezogen werden sollen. All diese Prozesse haben bis zu diesem Zeitpunkt die Audit-ID Null.

Normalerweise wird die Event-Maske und die Audit-Struktur eines Prozesses beim Login eines Nutzers initialisiert, da diese Prozesse bereits vorhanden sind und nicht aus einem Login-Prozess hervorgegangen sind, muss die Initialisierung der notwendigen Strukturen mittels des Kommandos '`/usr/sbin/aca -s -c "login 0"`' nachträglich erfolgen. Das Steuerungsprogramm simuliert mit diesem Kommando den Login des Nutzers mit der `uid` 0. Infolgedessen wird für jeden Prozess, die Event-Maske gesetzt und eventuell vorhandene ACEs eingefügt. Außerdem wird sichergestellt, dass die Audit-ID für die bereits laufenden Prozesse nicht mehr unautorisiert verändert werden kann.

Schritt 5:Initialisieren der Capabilities

Die Initialisierung der Capabilities muss im init-Skript erfolgen, da dies nach einem eventuellen Wechsel in der **`secure`**-Mode nicht mehr möglich ist. Beim Einlesen der Konfigurationsdatei wird lediglich das Vorhanden sein von Capabilities überprüft und die Startzeile ermittelt. Das Einlesen der Capabilities erfolgt erst in diesem Schritt.

Jede Regel besteht aus dem Typangabe (`aid` oder `prog`), der Audit-ID und der Capability (**`admin`**, **`setaid`** oder **`msg`**). Die einzelnen Schlüsselwörter müssen durch ein Leerzeichen getrennt sein und in einer Zeile stehen. Befindet sich am Ende der Zeile ein Backslash, ist eine weiter Capability in der nächsten Zeile angegeben. Das Setzen der Capabilities erfolgt mittel des Steuerungsprogramms. Beispielsweise wird das Kommando '`/usr/bin/aca -s -c "setcap aid: 101 msg"`' zum Einfügen der Capabilities `msg` für die Audit-ID 101 ausgeführt. Die Kommandos werden teilweise aus den Werten der Konfigurationsdatei zusammengebaut, sind diese Werte inkorrekt, schlägt der Aufruf fehl. Aufgrund des Parameters `-s` erfolgt keine Fehlerausgabe.

Das Einfügen einer Capability ist nicht davon abhängig, ob ein Prozess mit der entsprechenden Audit-ID existiert.

Schritt 6:Erzeugen der Wrapper-Skripte

Ein Großteil des Sicherheitskonzeptes des Audit-Systems ist von der Audit-ID abhängig. Dämon-Prozesse, die während der Startphase erzeugt werden, besitzen in der Regel die Audit-ID 0 und sind somit weder voneinander zu unterscheiden noch ist es möglich, den einzelnen Prozessen gewisse Rechte innerhalb des Audit-Systems zu zuteilen. Da die Identifizierung der Prozesse (initiierte Events) und die Abtestung der Capabilities anhand der Audit-ID erfolgt, muss eine Möglichkeit bestehen Dämon-Prozessen eine eigene Audit-ID zu geben.

Ist in der Konfigurationsdatei der Parameter `app_wr` auf den Wert `script` gesetzt, wird die Datei `/etc/security/audit_aid.conf` eingelesen und für jedes darin aufgeführte Programm oder Skript ein sogenanntes Wrapper-Skript erzeugt (siehe 3.1.4).

Schritt 7:Starten des Audit-Dämon

Sollte der Parameter `boot` auf den Wert `run` gesetzt sein, wird in diesem Schritt der Audit-Dämon gestartet. Dies muss vor einem eventuellen Wechsel in den **`secure`**-Mode erfolgen, da es unter Umständen notwendig ist, die Audit-ID des Dämon-Prozesses zu setzen. Hierzu wird zunächst mittels des Kommandos '`aca -s -c getadmin`' die Audit-ID des aktuellen Audit-Administrators ermittelt. Ist diese verschieden von 0, muss die Audit-ID mittel des Applikation-Wrappers entsprechend gesetzt werden. Ansonsten ist es dem Audit-Dämon während des Betriebs unmöglich die notwendigen Systemrufe auszuführen.

Schritt 8:Wechseln der Audit-Runlevels

Als letzte Aktion wird das Audit-System in den richtigen Betriebsmodus überführt. Damit eine Konfigurierung des Audit-Moduls überhaupt möglich ist, muss sich das System im **`normal`**-Mode befinden. Wird das init-Skript nach dem Wechsel in den **`secure`**-Mode aufgerufen, ist ein Großteil der notwendigen Operationen unmöglich und es erfolgt die Ausgabe einer Reihe von Fehlermeldungen.

4.5.2 Applikation-Wrapper

In den bisherigen Abschnitten wurde bereits mehrfach die Verwendung des Applikation-Wrappers `apwr` erwähnt. Hierbei handelt es sich um ein Programm, welches eine Audit-ID und eine ausführbare Datei mit den notwendigen Argumenten als Parameter erwartet. Der Applikation-Wrapper setzt die Audit-ID des eigenen Prozesses auf den übergebenen Wert und überlagert sich anschließend mittels des Systemrufes `execve(...)` mit dem ihm übergebenen Programm. Damit läuft des soeben gestartete Programm mit der gewünschten Audit-ID.

Die Ausführung des Programms ist nur erfolgreich, wenn der aktuelle Prozess in der Lage ist sein Audit-ID zu verändern.

4.6 Ressourcenverbrauch

Das Audit-System arbeitet abgesehen vom Konfigurationsprogramm vollständig im Hintergrund und sollte den normalen Betrieb eines produktiven Systems nicht behindern. In diesem Abschnitt wird kurz auf den Ressourcenverbrauch des Audit-Systems eingegangen.

4.6.1 Vergleich der allgemeinen Systemleistung

Ein wichtiger Punkt ist die allgemeine Systembelastung. Zu diesem Zweck wurden drei verschiedene Tests durchgeführt. Die Messungen erfolgten auf den folgenden Testsystemen:

1. Dual Pentium I MMX 233 MHz (mit gemeinsamen Cache 512kByte)
   192 MB Speicher
   Adaptec UltraWide SCSI-System

2. Pentium III (Katmai) 450 MHz
   64 MB Speicher
   IDE-System

Bei den Tests wurden alle Audit-Events protokolliert und eine geringe Anzahl von ACEs eingefügt. Im späteren Einsatz sollten weitaus weniger Audit-Daten generiert werden und die Anzahl der verwendeten ACEs sollte nur geringfügig größer sein. Es kann also davon ausgegangen werde, dass die hier ermittelten Wert der Praxis entsprechen sollten.

Übersetzen des Kernel Version 2.3.18 mit einer CPU

Hierbei handelt sich um den LOSA-Kernel, der sich aus, welchen Gründen auch immer, auf dem Dual-System nicht mit dem Kommando `make -j2 bzImage`, übersetzen lies. So dass das Übersetzen des Kernels nur auf einem Prozessor erfolgte. Der andere Prozessor sollte prinzipiell für die Generierung der Audit-Daten frei bleiben. Ansonsten liefen auf dem Rechner keine zusätzlichen Prozesse. Das 2. Testsystem besitzt nur eine CPU, so dass das Übersetzen des Kernels den Rechner zu 100 Prozent auslasten sollte.

Das Messen der Übersetzungszeit erfolgt mittels des Systemkommandos `date`, womit der Messvorgang wie folgt angestoßen wurde:

```
date > /tmp/start; make [-j2] bzImage; date > /tmp/stop
```

Die Differenz zwischen der Start- und der Stopzeit ergibt die Übersetzungszeit des Kernel. Da die Shell alle Kommandos sequenziell abarbeitet, ist auch auf einen SMP-System sichergestellt, dass das Kommando `date` erst nach dem vollständigen Übersetzen des Kernel aufgerufen wird.

Sowohl mit aktiviertem Audit als auch ohne Audit benötigt das Dual-System 14 min und das PIII-System 5 min zum Übersetzen des Kernel, d.h. das Generieren der Audit-Daten erzeugte keinen hier messbaren Overhead.

Übersetzen des Kernel Version 2.2.12 mit zwei Prozessoren

Die Kernel Version 2.2.12 lies sich mit beiden Prozessoren übersetzen, so dass auch das Dual-System zu 100 Prozent ausgelastet werden konnte. Jedoch war auch hier kein Unterschied messbar, sowohl bei aktiviertem als auch bei abgeschaltetem Audit benötigte der Rechner 10 min für das Übersetzen des Kernels.

D.h. auf einem System, welches zum Großteil mit Rechenoperationen beschäftigt ist, ist der Overhead des Audit-Systems nicht messbar. Das gilt sowohl für SCSI- als auch für IDE-Systeme, bei denen die Festplattenzugriffe zum Großteil vom Hauptprozessor gesteuert werden.

Starten der X-Window Oberfläche

Das Starten der X-Window Oberfläche erfordert zum Großteil Festplattenzugriffe und ist weniger rechenintensiv. Um den Effekt, das Linux Caches zu entgehen wurde `startx` zweimal hintereinander ausgeführt.

Für das Starten der X-Window Oberfläche benötigte das Dual-System ohne Audit beim ersten Start 24s und beim zweiten Start 14s. Wurde das Audit-System aktiviert, dauerte der erste Start ebenfalls 24s und der zweite Start 16s. D.h. auch hier konnten keine wirklichen Unterschiede festgestellt werden.

4.6.2 Menge der anfallenden Audit-Daten

Bei der Installation des Audit-Systems sollte darauf geachtet werden, dass genügend Festplattenplatz für die anfallenden Audit-Daten zur Verfügung steht. Bei den zuvor beschriebenen Tests wurden alle Audit-Events protokolliert. Dies sollte nicht der Normalfall sein, zeigt aber viele Audit-Daten in kürzester Zeit generiert werden können.

Testlauf	Menge der Audit-Daten
Übersetzen des Kernels	36 MB
Starten der X-Window Oberfläche	3 MB

Tabelle 4 : Menge der Audit-Daten bei den Testläufen

Bei der Implementierung und bei späteren Tests wurde das Audit-System in einer relativ einsatznahen Konfiguration betrieben. Der Menge der erzeugten Audit-Daten war hier besonders von der Art der Aufgabenstellung, die gerade auf dem Rechner bearbeitet wurde, abhängig. Bei regulären Büroarbeiten, dies umfasst das Schreiben von Texten und das gelegentliche Starten von Anwendungen, wurden an einem 8 Stunden Arbeitstag ca. 2-4 MB Audit-Daten erzeugt. Handelte es sich bei den Tätigkeiten um Programmieraufgaben, stiegt die Menge der Audit-Daten schnell auf über 40 MB pro Tag an.

Alle Messungen wurden auf einem Single-User-System durchgeführt, d.h. die Angaben bezogen sich auf einen Nutzer. Auf einem Applikationsserver ist das Datenaufkommen entsprechend der Anzahl der Nutzer höher, so dass bei 20 Nutzern über 200 MB Audit-Daten pro Tag nicht ausgeschlossen werden können.

Kapitel 5

Zusammenfassung

5.1 Vergleich der vorgestellten Konzepte

Die Grundidee von LOSA lag in der Erstellung eines Betriebssystem-Audit, ähnlich dem BSM von Solaris in Kombination mit einigen ausgewählten Eigenschaften von Windows NT. Dieser Abschnitt soll einen Überblick über die Leistungsfähigkeit der Konzepte geben und zeigen, wo LOSA einzuordnen ist.

5.1.1 Aufzeichnungsumfang

Der Aufzeichnungsumfang ist die wichtigste Eigenschaft eines Betriebssystem-Audit. Er beschreibt direkt die Leistungsfähigkeit des Audit-Systems. Hierbei ist nicht nur die Menge der aufgezeichneten Ereignisse von Bedeutung, sondern auch die Menge der Informationen, die ein Ereignis beschreiben.

	Windows NT	Solaris	Linux
Identifikation d. Initiators	- Nutzer-ID- - Rechner ID auf welchem die betroffene Ressource liegt	- Audit-ID - Session-ID - IP-Adresse des Herkunftsrechners	- Audit-ID - initiale Gruppe - Session-ID - IP-Adresse des Herkunftsrechners
Dateisystemoperationen	Alle	Alle	- alle dateiverändernde Operation - kein read/write Operationen
- Ressourcenangabe	- Dateiname	- Dateiname - absoluter Pfad	- Dateiname - absoluter Pfad
- Dateiinformation	- Besitzer	- Inode - Gerät - Besitzer - Zugriffsrechte	- Inode - Gerät - Besitzer - Zugriffsrechte
Netzwerk	Keine	Alle Socket- bzw. Streamoperationen	Verbindungsaufbau - connect(...)/accept(...)
Prozesssystem	Alle	Alle	alle systemverändernde Operationen
IPC	Keine	Alle	Keine
Systemstart- und stop	Beide	Beide	Nur Start
Login und Logout	Beide	(Beide)	Nur Login
Offen für andere Applikation	Ja	Ja	Ja

Tabelle 5 : Vergleich des Aufzeichnungsumfangs

Wie die Tabelle 5 zeigt ist die Menge, der von den einzelnen Systemen aufgezeichneten Informationen, nahe zu gleich. Im Gegensatz zu den im Abschnitt 2.2 erwähnten Audit-Projekten für Linux ist LOSA in der Lage eine ähnliche Menge von Informationen zu sammeln.

Allerdings ist der Aufzeichnungsumfang immer noch geringer als beim BSM von Solaris. Dies liegt nicht an einer schlechten Konzeptionierung oder Implementierung von LOSA, sondern liegt daran, dass bei der Auswahl der aufzuzeichnenden Systemrufe bewusst ganze Gruppen von Systemrufen (IPC, Datei read/write) weggelassen wurden. Erfahrungen mit den Systemen von Sun Microsystems und Microsoft haben gezeigt, dass diese Daten zu umfangreich und für eine sicherheitskritische Analyse nicht aussagekräftig sind und somit eine Aufzeichnung nicht lohnenswert ist.

Außerdem wird deutlich, dass LOSA nicht in der Lage ist, das Abmeldungen von Nutzern oder das Herunterfahren des Systems zu protokollieren. Dies liegt an der Vielfalt von Möglichkeiten zur Initiierung eines solchen Events. Da man für ein Audit möglichst die initiale Aktion benötigt, ist die Protokollierung solcher Ereignisse auf einem UNIX-System extrem schwierig. Aus dem selben Grund wird wohl auch SUN von Microsystems beim BSM auf diese Events verzichtet haben. Das Herunterfahren von Linux ist eigentlich nur mit dem Systemruf `reboot(...)` möglich. Jedoch kehrt dieser Systemruf nicht mehr zurück, sodass das Ausführen von Dateioperationen, die für die Protokollierung notwendig sind, mehr möglich ist.

5.1.2 Konfigurierungsmöglichkeiten

Als zweites Kriterium werden wie im Abschnitt 2.3 die Filter- und Konfigurierungsmöglichkeiten der einzelnen Systeme verglichen. LOSA ist für ein UNIX-System konzipiert und besitzt somit wie das BSM für Solaris keine graphischen Benutzeroberflächen zu Konfigurierung des Systems. Die Konfigurierung erfolgt ausschließlich mittels der beschriebenen Konfigurationsdateien und des Steuerungsprogramms. Windows NT bietet die Möglichkeit alle Einstellungen in graphischen Dialogfenstern vorzunehmen.

	Windows NT	Solaris	Linux
Konfigurierung erfolgt mittels	- einzelner Dialogfelder	- globaler Skripte - Managementsoftware	- globaler Skripte - Managementsoftware
Rekonfigurierung zur Laufzeit	Uneingeschränkt möglich	Eingeschränkt möglich	Uneingeschränkt möglich, es sei denn das System befindet sich im secure-Modus
Aufzeichnungs-granularität			
Unterteilung der Event	- Einzelne Events	- 32 Event-Klassen	- 32 frei definierbare Event-Klassen - einzelne Event
Verknüpfen mit	- Systemweit	- theoretisch nur Prozess, jedoch mittels Login auch Nutzer und Gruppen	- Gruppen - Nutzer - Prozesse
Ressourcenbezogen	Ja	Nein	Ja
Trennung der Privilegien	Ja (durch den Sicherheitsoperator)	Nein	Ja (durch die Vergabe von Capabilities)
Trail			
- einzelne Trails	Nein	Ja	Ja
- freies Zielverzeichnis	Nein	Ja	Ja
- Wechsel zu Laufzeit	Nein	Ja	Ja
- Trail-Limits	Nein	Nein	Ja
Verschiedene Betriebsmodi	Nein	Nein	Ja drei

Tabelle 6 : Vergleich der Konfigurationsmöglichkeiten

Es bleibt sicherlich eine Sache des Betrachters, welche Kriterien zur Bewertung eines Audit-Systems herangezogen werden. Anderseits haben sich die in Tabelle 6 aufgeführten Kriterien als sinnvolle

Vergleichspunkte und als notwendige Eigenschaften eines Audit-Systems erwiesen. Die Implementierung von LOSA erfüllt nahezu alle genannten Punkt. Dies ist darauf zurückzuführen, dass bei der Konzeptionierung gerade diese Punkte als zu implementierende Eigenschaften vorgesehen wurden. Bis auf die Betriebsmodi, die nur im LOSA-Projekt implementiert sind, sind alle anderen Eigenschaften entweder im Windows Audit oder im BSM von Solaris enthalten. Beiden Konzepten fehlen Eigenschaften, die sich bei der Nutzung des jeweils anderen Systems, als nützlich bzw. vorteilhaft erwiesen haben. Bei LOSA wurde versucht, genau diese Lücken zu schließen.

5.2 Was bleibt offen?

Nachdem im vorherigen Abschnitt die guten Eigenschaften von LOSA beschrieben wurden, soll dieser Abschnitt dazu dienen, einige offene Arbeitspunkte zu nennen. Sowohl bei der Implementierung als auch bei der Auswertung des entstandenen Paketes haben sich verschiedene Punkte gezeigt, die von einer Überwachung ausgeschlossen sind oder eine sinnvolle Erweiterung darstellen würden.

Unterstützung von Samba

Bis heute ist Linux eigentlich kein System für den Heimanwender, sein Haupteinsatzgebiet liegt im Server-Segment und dies hat es nicht zuletzt dem Samba-Projekt zu verdanken. Mittels Samba ist es möglich, einen Rechner mit Linux als Server für Windowsnetzwerke zu betreiben. Der Linux-Server in Verbindung mit Samba kann nahezu alle Operationen übernehmen, die in der Windows NT Server Version implementiert sind. Hierzu gehören alle Dateioperation, die Bereitstellung eines Druckdienstes und seit der Version 2.0.x die Verwendung als primären Domänen-Controller (PDC).

Auf dem Linux-Server werden all diese Dienste zum Samba-Dämon erbracht, welcher in der Regel von root gestartet wird. Für LOSA werden somit alle Datei- und Druckeroperationen vom Nutzer root ausgeführt. Zwar ist es möglich dem Samba-Dämon eine eigene Audit-ID zu geben, jedoch kann damit nur der Dämon von allen anderen root-Prozess getrennt werden. Was man benötigt, ist eine Erweiterung des Samba-Paketes um die notwendigen Audit-Funktionen. So dass der Nutzer identifiziert werden kann, welcher die Operationen ausgeführt hat.

Automounter

LOSA überwacht den Systemruf `mount()`, damit kann jede mount-Operation, die mittels des Programms **`mount`** ausgeführt wurde, überwacht werden. Jedoch besteht seit der Kernel-Version 2.2.x die Möglichkeit, Dateisysteme dynamisch zu mouten, wenn diese benötigt werden. Dies geschieht im Kernel ohne die Verwendung des Systemrufes `mount()` und somit von LOSA unbemerkt. Damit ist es prinzipiell möglich, Dateisysteme einzubinden, ohne das dies später genau nachvollzogen werden kann.

NFS-Operationen

Ähnlich wie Samba dient NFS (Network File System) zum Exportieren von Dateisystemen an entfernte Rechner und wird von einem Dämon gesteuert. Allerdings ist der NFS-Dämon nur noch der Start-Prozess zur Bereitstellung des Dienstes. Alle vom entfernten Rechner ausgeführten Dateioperationen verlassen den Kernel gar nicht mehr. Sie werden direkt vom Netzwerk-Stack an die nfs-Funktionen übergeben und dort ausgeführt. Für jede nfs-Operation existiert eine zugehörige Funktion (`nfs_open()`, `nfs_write()`, `nfs_read()`, ...).

Da LOSA nur die Operationen des virtuellen Dateisystems überwacht, bleiben alle NFS-Operationen auf dem Server unbemerkt. Läuft auf dem nfs-Client keine Audit, besteht keine Möglichkeit, eine ausreichende Protokollierung des Dateioperationen zu gewährleisten.

Host-basiertes Netzwerk-Audit

In der Einleitung wurde bereits auf ein Netzwerk-Audit eingegangen, hierbei handelt es sich um die Überwachung von Netzwerk-Operationen. Zwar werden im LOSA-Projekt die Systemrufe `accept()` und `connect()` aufgezeichnet, jedoch belegen diese nur den Aufbau einer regulären Verbindung. Bis zu diesem Punkt werden allerdings eine Vielzahl von Netzwerk-Operationen

ausgeführt, die an sich oder in einer bestimmten Reihenfolge eine eindeutige Sicherheitsverletzung darstellen können.

Um diese zu protokollieren, bedarf es einiger Erweiterungen im Netzwerk-Stack. Die Erweiterungen sind nicht umfangreicher bzw. komplizierter, als die Erweiterungen des LOSA-Projekt in den eigentlichen Systemrufen. Viel umfangreicher und komplizierter ist Implementierung einer geeigneten Infrastruktur zur Verarbeitung der anfallende Daten. Der Audit-Puffer im LOSA-Projekt ist so implementiert, dass er verschiedene Audit-Record-Formate unterscheiden kann, damit bietet LOSA eine grundlegende Infrastruktur zur Umsetzung eines Host-basierten-Netzwerk-Audit für Linux.

Anhang A
Systemintegration

A.1 Hardware-Voraussetzungen

Im Abschnitt 4.6 wurde bereits erwähnt, dass LOSA nur einen geringen Overhead erzeugt, sodass für den Betrieb des Audit-Systems keine besonderen Hardware-Anforderungen notwendig sind. Einzig ausreichend Speicherplatz für die anfallenden Audit-Daten muss zur Verfügung gestellt werden.

Es ist allerdings zu beachten, dass LOSA bisher **nur für Intel-basierte System verfügbar** ist. Da es unumgänglich wurde Hardware-spezifischen Code anzupassen, muss dieser Teil erst auf die jeweilige Zielplattform portiert werden.

A.2 Software- Voraussetzungen

Das LOSA-Projekt wurde auf einer Helloween-Distribution implementiert. Alle notwendigen Pakete wurden dieser Distribution entnommen. Da die Helloween ist ein Lizenz-Produkt von RedHat ist, enthält die RedHat-Distribution Version 6.0 die gleichen Pakete. S.u.S.E verwendet eine geringfügig andere Verzeichnisstruktur und basiert teilweise auf anderen Quell-Code-Paketen. Aus diesem Grund ist zur Zeit kein vollständiges LOSA-Paket für S.u.S.E verfügbar.

Mit etwas Erfahrung beim Übersetzung und Einrichten von Software-Paketen, kann die Installation von LOSA auch auf jeder anderen Distribution durchgeführt werden. Einzig die im folgenden aufgeführten Software-Pakete sollten vorhanden sein.

- Kernel:
  - linux-2.3.18
  - (linux-2.4.0-test1, ..-test3, ..-test6)
- Compiler:
  - egcs-2.91.66
  - yacc
  - lex-2.5.4
- Quelle-Code:
  - util-linux-2.9w.tar.gz
  - sh-utils-2.0.tar.gz
  - kdebase-1.1.1.tar.gz
- Für die Verwendung des Trail-Viewers:
  - Tcl8.0 and Tk8.0
    - Wichtig, neuere Versionen von Tcl/Tk arbeiten nicht mit dem Tix-Paket zusammen.
  - Tix4.1.8

A.3 Installation

Die Installation von LOSA kann sowohl per Hand als auch mittels eines install-Skriptes erfolgen. Allerdings ist lediglich ein install-Skript für RedHat verfügbar. Da die von S.u.S.E verwendeten Systemapplikationen noch nicht angepasst wurden, ist das vorhandene Skript noch unvollständig. Im folgenden wird sowohl die Installation mittels des Skriptes als auch per Hand erläutert.

Für die Installation sollte in jedem Fall fundierte Kenntnisse über Linux und dessen Kernel-Konfigurierung vorhanden sein. Es ist nicht empfehlenswert LOSA zu installieren, wenn man noch nie den Kernel oder ein anderes Programm auf einem Linux-System übersetzt hat. (Bei einer falschen Installation oder Konfiguration kann es unter Umständen recht knifflig werden, dass System wieder in den Originalzustand zurück zuversetzen.)

Es ist auf jeden Fall empfehlenswert, den boot-Loader so zu konfigurieren, dass im Notfall ein anderer Kernel gestartet werden kann. Sollte während der Installation oder beim ersten Restart des Systems ein Fehler auftreten, kann der Rechner notfalls mit dem alten Kernel gestartet werden. Da die LOSA-Installation auch einige Eingriffe in die Startkonfiguration des Systems erfordert, sollte der Rechner im Fehlerfall nur im single-user-mode gestartet werden.

Zum Installieren von LOSA sind zwingend root-Privilegien erforderlich.

A.3.1 Install-Skript

Zur Installation von LOSA mittels eines Skriptes werden die Dateien `install_redhat.sh` und `install_suse.sh` mitgeliefert. In den ersten Zeilen der Dateien, erfolgt die Konfiguration die Installation, dabei haben die Variablen folgende Bedeutung:

`user`: Besitzer der ausführbaren Dateien,
`group`: Gruppe des Besitzers der ausführbaren Dateien,
`manuser`: Besitzer der man-Pages,
`mangroup`: Gruppe des Besitzers,
`libuser`: Besitzer der Bibliotheken,
`libgroup`: Gruppe des Besitzers der Bibliotheken,

`LINUXVERSION`: verwendete Kernel-Version (z.B. 2.4.0-test1),
`LINUX`: kompletter Name des Kernel-Source-Verzeichnisses (z.B. linux-2.4.0-test1),
`LINUX_SRC`: Pfad zum Kernel-Source-Verzeichnis (z.B. `/usr/src`).

Package: `su`, `login`, `kdm`
`install_[package]`: gleich 1 wenn `[package]` installiert werden soll,
`[package]`: der Name der `[package]`-Sourcen (z.B. für login `util-linux-2.9w`),
`[package]_SRC`: der Pfad zu den `[package]`-Sourcen,

`backup_dir`: Backup-Verzeichnis für die Installation,
`audit_dir`: Verzeichnis für die Audit-Trails (vgl. `auditd.conf`),

`PREFIX`: Verzeichnis für die Binaries,
`LIBPREFIX`: Verzeichnis für die Bibliotheken,
`INCPREFIX`: Verzeichnis für die Header der Bibliotheken,
`MANDIR`: Verzeichnis für die man-Pages,

`with_backup`: gleich 1, wenn ein Backup von den zu patchenden Paketen erzeugt werden soll,
`runlevels`: die Runlevels in denen das Audit-Modul gestartet werden soll

Wurden alle Variablen des install-Skriptes angepasst, sollten noch die Dateien

`patches/linux-2.4.0-test1/kpatch.sh`: install-Skript des Kernel-Patches,
`man/install_script.sh`: install-Skript für die man-Pages,
`scripts/audit`: init-Skript des Audit-Moduls

überprüft werden, ob die dortigen Pfade der eigenen Konfiguration entsprechen.

Anschließend kann das install-Skript gestartet werden. Im ersten Durchlauf wird der Kernel gepatcht und das Vorhandensein der notwendigen Pakete überprüft. War dieser Durchlauf erfolgreich, ist mittels

```
make menuconfig
```

der Kernel zu konfigurieren und anschließend mittels

```
make dep clean bzImage modules modules_install
```

neu zu übersetzen. Beim Konfigurieren des Kernels muss das Übersetzen des Audit-Kernels aktiviert werden. Standardmäßig ist die Option `'Audit support'` im Menü `'Processor type and features'` auf `'no'` gesetzt.

Wurde der Kernel erfolgreich übersetzt, kann das install-Skript erneut aufgerufen werden. Abschließend sind lediglich noch die aktualisierten Pakete der Systemapplikationen neu zu übersetzen und die Konfigurierung des Audit-Modus vorzunehmen. Eine genaue Beschreibung hierzu erfolgt im Abschnitt der manuellen Installation.

A.3.2 Manuelle Installation

In diesem Abschnitt erfolgt die Erläuterung der manuellen Installation. Diese Variante LOSA zu installieren, ist immer dann notwendig, wenn eine der Bedingungen für die automatische Installation nicht erfüllt ist und insbesondere bei Systemen auf denen eine andere Distribution als RedHat installiert ist.

Patchen des Kernels

Das Patchen des Kernels erfolgt mittels des im Verzeichnis `patches/linux-[Version]-patch/` vorhandenen Skriptes. Das Skript erstellt im Kernel-Verzeichnis ein Verzeichnis namens `audit`, in welches alle Dateien des Audit-Moduls kopiert werden. Zudem werden die notwendigen Header im Verzeichnis `include/linux` abgelegt. Anschließend wird der Kernel-Patch eingespielt.

Wurde in den Kernel bereits ein anderer Patch eingespielt oder entspricht die verwendete Kernel-Version nicht der Patch-Version, kann das Installieren des Patches bei einigen Dateien fehlschlagen. In diesem Fall müssen die betroffenen Datei per Hand nacheditiert werden. Das Programm **`patch`** hinterlegt für jede Datei, die nicht aktualisiert werden konnte, zwei Dateien namens `[orig_name].orig` und `[orig_name].rej`. Die Datei `*.rej` enthält die nicht eingefügten Zeilen des Patches. Diese Zeilen müssen dann nachträglich per Hand eingefügt werden. Hierzu empfiehlt es sich, einen ungepatchten Kernel bzw. einen Kernel der richtigen Version zu patchen und diesen als Referenz zu verwenden.

Damit das Audit-Modul beim Übersetzen des Kernels eingebunden wird, muss dieses in der Kernel-Konfiguration aktiviert sein. Nach der Installation des Patches ist dies nicht der Fall, standardmäßig ist die Option `'Audit support'` im Menü `'Processor type and features'` auf `no` gesetzt.

Bevor die nächsten Schritte der Installation ausgeführt werden können, muss der Kernel mittels:

```
make dep clean bzImage modules modules_install
```

konfiguriert und übersetzt werden.

Installieren der Audit-Bibliothek

Die Installation der Audit-Bibliothek erfolgt mittels `make` und `make install` innerhalb des Verzeichnisses `libaudit`. Hierbei ist unbedingt darauf zu achten, dass die Version des verwendeten Kernel-Patches mit der Version der Audit-Bibliothek übereinstimmt. Insbesondere bei den Version 2.3.18 und 2.4.0-test1 wurden entscheidene Änderungen vorgenommen, so dass beide Version nicht kompatibel sind.

In der Datei `Makefile.global` können die Pfade und Besitzer der Dateien der Bibliothek fetsgelegt werden. Normalerweise wird die Bibliothek in das Verzeichnis `/usr/lib` installiert, sollte dies nicht gewünscht sein, kann im Makefile ein anderes Verzeichnis angegeben werden. Da alle LOSA-Applikationen auf die Audit-Bibliothek zugreifen, ist darauf zu achten, dass das neue Verzeichnis in den Environment-Variablen `LD_LIBRARY_PATH` und `LD_RUN_PATH` enthalten oder in den Makefiles der LOSA-Applikationen zusätzlich angegeben ist.

Schlägt das Übersetzen der Bibliothek fehl, so liegt dies meist daran, dass der Kernel noch nicht übersetzt bzw. konfiguriert wurde und damit der Link für das `include`-Verzeichnis noch nicht gesetzt wurde.

Installieren der audit-Applikationen

Die einzelnen Applikationen befinden sich in den Unterverzeichnissen des Verzeichnis `utils`. Für den Betrieb von LOSA sind die Applikationen:

- **`aca`**: Audit Control Application – Managment-Programm,
- **`auditd`**: Audit-Dämon,
- **`apwr`**: Applikation-Wrapper und
- **`aclctl`**: ACL Controll – ACL-Datenbank Generator

zwingend erforderlich. Das Programm:

- **`atv`**: Audit Trail-Viewer

ist lediglich für die Betrachtung der Audit-Daten notwendig. Sollte aber, so kein anderes Programm vorhanden ist, installiert werden.

Die Installation kann entweder für jedes Programm separat aus dem jeweiligen Verzeichnis oder direkt aus dem Verzeichnis `utils` heraus erfolgen. Wie bei der Audit-Bibliothek werden die Pfade und die Nutzerrechte in der Datei `Makefile.global` festgelegt. Dies gilt sowohl für die separate als auch für die gemeinsame Installation der Programme.

Installation der man-Pages

Für die Installation der man-Pages befindet sich im Verzeichnis man das Skript `install_man.sh`. Hierbei handelt es sich um ein Shell-Skript, welches die Installation der einzelnen Pages in die jeweiligen Unterverzeichnisse übernimmt. Die im Skript vorhandenen Variablen

- `USR`: Besitzer der man-Pages (default: `root`),
- `GRP`: Gruppe des Besitzers (default: `man`) und
- `MANDIR`: Installationsverzeichnis (default: `/usr/man`)

können, so dies notwendig ist, entsprechend der Richtlinien des eigenen Systems angepasst werden.

Installation der Konfigurationsdateien

Im Verzeichnis `config` befinden sich die Konfigurationsdateien des Audit-Modus. Standardmäßig werden die Dateien in das Verzeichnis `/etc/security` kopiert. Sollte dies nicht den allgemeinen

Richtlinien des Systems entsprechen und ein anderes Zielverzeichnis festgelegt werden, ist darauf zu achten, dass alle Applikationen ihre Konfigurationsdateien in diesem Verzeichnis suchen. Um die Programme des Paketes nutzen zu können, muss das neue Verzeichnis jedem Programm als Parameter übergeben werden.

Im Anhang D befindet sich eine Beschreibung der mitgelieferten Konfiguration, normalerweise entspricht diese Konfiguration den allgemeinen Anforderung zum Testen des Audit-Systems. Lediglich die Nutzer-IDs sollten angepasst werden. Für eine weitergehende Konfigurierung des Systems sollten die Beschreibungen aus dem Abschnitt 3.1.4 und dem Anhang C.3 hinzugezogen werden.

In der Datei `auditd.conf` wird unter anderem das Verzeichnis Audit-Dateien und das Verzeichnis für die `connect`-Datei festgelegt. Bevor das Audit-Modul aktiviert werden kann, müssen diese Verzeichnisse erstellt werden. In der vorgegebenen Konfiguration sind dies die Verzeichnisse `/audit` und `/var/run`. Für den Einsatz auf einem produktiven System ist es ratsam, wenn das Verzeichnis der Audit-Dateien auf eine separate Partition verweist, dies verhindert, dass wichtige Partitionen vollgeschrieben werden.

Installation der Skript-Dateien

Für den Betrieb des Audit-Moduls sind die vier Skripte

- `audit`: Audit-Init-Skript
- `audit_shutdown`: Shutdown-Skript
- `audit_warn`: Warn-Skript
- `awinit`: Applikation-Wrapper Skript

notwendig.

Das Skript `audit` sollte in das Verzeichnis der init-Skripte des Systems kopiert werden. Bei RedHat und bei fast allen anderen Linux-Distributionen ist dies das Verzeichnis `/etc/rc.d/init.d`. Anschließend muss noch ein Link im gewünschten Runlevel-Verzeichnis erzeugt werden. Das aktuelle Runlevel sollte in der Datei `/etc/inittab` zufinden sein. Werden mehrere Runlevel verwendet, sollte in den jeweiligen Verzeichnis ebenfalls ein Link erzeugt werden. Das Erzeugen des Links erfolgt mittels

```
ln -s ../init.d/audit S01audit
```

aus dem Zielverzeichnis heraus. Es ist ratsam dem Audit-Skript die Nummer S01 zugeben, dadurch wird der Audit-Dämon vor allen anderen Programmen gestartet und kann diese damit von Anfang an überwachen. In jedem Fall muss das init-Skript vor den Skripten, die mittels des Applikation-Wrappers verändert wurden, gestartet werden, anderenfalls können wichtige Einstellungen und damit Audit-Daten verloren gehen.

Die Skripte `audit_shutdown` und `audit_warn` sollten entsprechend der Konfiguration in die jeweiligen Verzeichnisse kopiert werden. Bei der vorgegebenen Konfiguration ist dies das Verzeichnis `/etc/security`.

Das Wrapper-Init-Skript wird standardmäßig in das Verzeichnis `/usr/sbin` installiert. Es wird entweder vom Administrator oder vom Audit-init-Skript zum Erzeugen der Wrapper-Skripte benötigt. Sollte in der Datei `audit.conf` die Variable `app_wr` auf `script` gesetzt sein, ist darauf zu achten, dass das init-Skript das Wrapper-Skript findet.

Wie bereits mehrfach erwähnt kann das Audit-Modul ohne root-Privilegien betrieben werden. In diesem Fall müssen die Skripte und alle von den Skripten aufgerufenen Programme von Audit-Administrator ausführbar sein.

Erstellen der ACL-Datenbank

Das Erstellen der ACL-Datenbank erfolgt mittels des Programms **aclctl**, welches zu diesem Zeitpunkt bereits installiert sein sollte. Bevor man das Programm startet, sollte die Konfigurationsdatei `audit_acl.conf` angepasst werden. Anschließend kann durch aufrufen des Programms **aclctl** ohne Parameter die erste ACL-Datenbank erzeugt werden.

Sollte bereits ein Datenbank vorhanden sein oder die Konfigurationsdatei nicht im Verzeichnis `/etc/security` liegen, muss das Programm mit den entsprechenden Parametern aufgerufen werden, bspw.:

```
aclctl -d -c /my_dir/my_acl.conf
```

Eine genauere Beschreibung ist in den Abschnitten 3.1.4 und 3.2.3 sowie im Anhang D.5 zu finden.

Patchen der Systemapplikationen

Für den korrekten Betrieb des Audit-Modus ist es zwingend notwendig, dass beim Anmelden eines Nutzers die Audit-Environment des Nutzer in den Kernel übertragen wird. Hierzu sind zwei Patches vorhanden. Zum einen für das Programm **login**, welches beim textuellen Login und vom inet-Dämon verwendet wird und ein Patch für das Programm **kdm**, welches einen graphischen Login bereitstellt.

Beide Programme müssen aus den Source-Paketen erstellt werden. Es ist empfehlenswert zunächst die Programme ohne die Audit-Patches zu übersetzen und zu testen. Beide Pakete müssen vor dem Übersetzen mittels des Skriptes `configure` konfiguriert werden und können anschließend, mittels **make** übersetzt werden. Eine Installation mittels make install ist nicht erforderlich, da die Pakete eine Vielzahl von Programmen enthalten, die nicht notwendig sind. Es reicht die beiden Dateien `kdm` und `login` in die entsprechenden Verzeichnisse zu kopieren.

Ist das Anmelden an das System mit den neuen Programmen möglich, können diese mittels

```
cd /usr/src/redhat/SOURCES
patch -p0 < [losa-package]/patches/[package]-losa.patch
cd [package]
make
```

gepatcht und neu übersetzt werden.

Außerdem existiert ein Patch für das Programm **su**. Beim Übersetzen des Programmes kann wie bei den Programmen **kdm** und **login** vorgegangen werden. Bei der Installation von **su** ist darauf zu achten, dass beim Kopieren des Programms das `setuid`-Recht verloren geht. Dies ist jedoch zwingend erforderlich und kann mittels

```
chmod u+s `which su`
```

neu gesetzt werden.

Anhang B

Überwachte Ereignisse

B.1 Systemrufe

In diesem Abschnitt erfolgt die Auflistung und Erläuterung der überwachten Systemrufe. Für jeden Systemruf wird die Datei, in welcher die zugehörige Funktion implementiert ist, und die Audit-Datei, in welcher die Audit-Funktion implementiert ist, aufgeführt. Außerdem wird dem Systemruf, die Event-Nummer und das zugehörige Define zugeordnet.

Wie bereits beschrieben, unterteilt sich der Audit-Record in einen statischen und einen variablen Teil. Der statische Teil wird immer gefüllt. In diesem Abschnitt wird für jeden Systemruf der variable Teil erläutert. Dabei handelt es sich um die Einträge `file` (Datei 1), `file2` (Datei 2), `env`, `arg1` (Arg 1) und `arg2` (Arg 2). Wird in `file` eine Datei, ein Verzeichnis, ein Device oder ein Link gespeichert, werden zwingend die Einträge `owner`, `gowner`, `dev`, `perm` und `inode` gefüllt. Im folgenden wird ein variabler Parameter nur aufgeführt, wenn ein Wert eingetragen wird.

sys_acct(const char *name)

- Kernel-Datei: `kernel/acct.c`
- Audit-Datei: `audit/audit_acct.c`

* Audit-Event: `AUE_ACCT(1)`
* variable Parameter:
  - Datei 1: `name`

sys_capset(cap_user_header_t header,const cap_user_data_t data)

- Kernel-Datei: `kernel/capability.c`
- Audit-Datei: `audit/audit_capability.c`

* Audit-Event: `AUE_CAPSET(2)`

sys_fork(struct pt_regs regs)
sys_clone(struct pt_regs regs)
sys_vfork(struct pt_regs regs)

- Kernel-Datei: `kernel/fork.c`
  `arch/i386/kernel/process.c`
- Audit-Datei: `audit/audit_process.c`

* Audit-Event: `AUE_FORK(3), AUE_CLONE(4), AUE_VFORK(44)`
* variable Parameter
  - Arg 1: `pid` den neuen Prozesses
* Besonderheiten: Alle drei Systemrufe sind in der Funktion `do_fork()` implementiert. Um in dieser Funktion das Event unterscheiden zu können, musste in der Funktion `do_fork(...)` ein zusätzlicher Parameter eingefügt werden.

sys_ioperm(unsigned long from, unsigned long num, int turn_on)

- Kernel-Datei: `arch/i386/kernel/ioport.c`
- Audit-Datei: `audit/audit_ioport.c`

* Audit-Event: `AUE_IOPERM(5)`

- variable Parameter
  - Arg 1: unteren 16 Bit `from`, oberen 16 Bit `num`
  - Arg 2: `turn_on`

sys_iopl(unsigned long unused)
  - Kernel-Datei: `arch/i386/kernel/ioport.c`
  - Audit-Datei: `audit/audit_ioport.c`
- Audit-Event: `AUE_IOPL(6)`
- variable Parameter
  - Arg 1: `unused`

sys_init_module(const char *name_user, struct module *mod_user)
  - Kernel-Datei: `kernel/module.c`
  - Audit-Datei: `audit/audit_module.c`
- Audit-Event: `AUE_INIT_MODULE(7)`
- variable Parameter
  - Datei 1: `name_user`
  - Datei 2: `mod_user->name` im Kernel vermerkter Name des Modules
- Besonderheiten: Die Funktion `sys_init_module(...)` ist mit Hilfe des Defines `CONFIG_MODULES` in der Datei `module.c` zweimal implementiert. Sollte die Modul-Unterstützung nicht aktiviert sein, gibt die Funktion immer den Fehler `ENOSYS` zurück.

delete_module(const char *name_user)
  - Kernel-Datei: `kernel/module.c`
  - Audit-Datei: `audit/audit_module.c`
- Audit-Event: `AUE_DELETE_MODULE(8)`
- variable Parameter
  - Datei 1: `name_user`
- Besonderheiten: siehe `sys_init_module(...)`

sys_link(const char * oldname, const char * newname)
  - Kernel-Datei: `fs/namei.c`
  - Audit-Datei: `audit/audit_namei.c`
- Audit-Event: `AUE_LINK(9)`
- variable Parameter
  - Datei 1: `oldname`
  - Datei 2: `newname`

sys_unlink(const char *pathname)
  - Kernel-Datei: `fs/namei.c`
  - Audit-Datei: `audit/audit_namei.c`
- Audit-Event: `AUE_UNLINK(10)`
- variable Parameter
  - Datei 1: `pathname`

sys_mknod(const char * filename, int mode, dev_t dev)
  - Kernel-Datei: `fs/namei.c`
  - Audit-Datei: `audit/audit_namei.c`
- Audit-Event: `AUE_MKNOD(11)`
- variable Parameter
  - Datei 1: `filename`
  - Arg 1: `mode`

- Arg 2: `dev`

`sys_rename(const char *oldname, const char *newname)`

- Kernel-Datei: `fs/namei.c`
- Audit-Datei: `audit/audit_namei.c`

- Audit-Event: `AUE_RENAME(12)`
- variable Parameter
  - Datei 1: `oldname`
  - Datei 2: `newname`

`sys_mkdir(const char *pathname, int mode)`

- Kernel-Datei: `fs/namei.c`
- Audit-Datei: `audit/audit_namei.c`

- Audit-Event: `AUE_MKDIR(13)`
- variable Parameter
  - Datei 1: `pathname`
  - Arg 1: `mode`

`sys_rmdir(const char *pathname)`

- Kernel-Datei: `fs/namei.c`
- Audit-Datei: `audit/audit_namei.c`

- Audit-Event: `AUE_RMDIR(14)`
- variable Parameter
  - Datei 1: `pathname`

`sys_symlink(const char *oldname,const char *newname)`

- Kernel-Datei: `fs/namei.c`
- Audit-Datei: `audit/audit_namei.c`

- Audit-Event: `AUE_SYMLINK(15)`
- variable Parameter
  - Datei 1: `oldname`
  - Datei 2: `newname`

`sys_open(const char *filename, int flags, int mode)`
`sys_create(const char *filename, int mode)`

- Kernel-Datei: `fs/open.c`
- Audit-Datei: `audit/audit_open.c`

- Audit-Event: `AUE_OPEN(16), AUE_CREAT(30)`

  Das Event `AUE_OPEN` wird lediglich zum Filtern verwendet. Im Audit-Record wird das Event noch einmal anhand des Argumentes `flags` unterschieden.

  `AUE_OPEN_R(18)` : read
  `AUE_OPEN_RC(19)` : read, create
  `AUE_OPEN_RT(20)` : read, truncate
  `AUE_OPEN_RTC(21)` : read, truncate, create
  `AUE_OPEN_W(22)` : write, truncate, create
  `AUE_OPEN_WC(23)` : write, create
  `AUE_OPEN_WT(24)` : write, truncate
  `AUE_OPEN_WTC(25)` : write, truncate, create
  `AUE_OPEN_RW(26)` : read, write
  `AUE_OPEN_RWC(27)` : read, write, create
  `AUE_OPEN_RWT(28)` : read, write, truncate
  `AUE_OPEN_RWTC(29)` : read, write, truncate, create

- variable Parameter
  - Datei 1: `filename`
  - Arg 1: `mode`
- Besonderheiten: Für die Systemruf `sys_creat(...)` existiert keine separate Audit-Funktion. Das Generieren des Audit-Records wird für diesem Systemruf durch die Funktion `audit_open(..)` erbracht. Normalerweise ruft `sys_creat(...)` mit festvorgebenden `flags` die Funktion `sys_open(...)` auf. Da beide Funktionen somit mittels `audit_open(...)` einen Audit-Record erzeugen würden, wurde die eigentlich `sys_open(...)`-Funktion nach `do_open(...)` umbenannt. Sie führt die ursprünglichen Aufgaben von `sys_open(...)` aus, erzeugt aber keinen Audit-Record. Die Funktion `sys_open(...)` wurde neu implementiert und initiiert mittels `audit_open(...)` das Generieren eines Audit-Events und ruft für die open-Operation die Funktion `do_open(...)` auf.

sys_uselib(const char * library)

  - Kernel-Datei: `fs/exec.c`
  - Audit-Datei: `audit/audit_open.c`
- Audit-Event: `AUE_USELIB(18)`
- variable Parameter
  - Datei 1: `library`
- Besonderheiten: Der Audit-Record wird wie beim Systemruf `sys_creat(...)` durch die Funktion `audit_open(...)` erzeugt.

sys_chdir(const char *filename)
sys_fchdir(unsigned int fd)

  - Kernel-Datei: `fs/open.c`
  - Audit-Datei: `audit/audit_open.c`
- Audit-Event: `AUE_CHDIR(31), AUE_FCHDIR(32)`
- variable Parameter
  - Datei 1: `filename`, bzw. der Dateiname hinter `fd`

sys_chmod(const char * filename, mode_t mode)
sys_fchmod(unsigned int fd, mode_t mode)

  - Kernel-Datei: `fs/open.c`
  - Audit-Datei: `audit/audit_open.c`
- Audit-Event: `AUE_CHMOD(33), AUE_FCHMOD(34)`
- variable Parameter
  - Datei 1: `filename`, bzw. der Dateiname hinter `fd`
  - Arg 1: `mode`

sys_chown(const char *filename, uid_t user, gid_t group)
sys_fchown(unsigned int fd, uid_t user, gid_t group)
sys_lchown(const char *filename, uid_t user, gid_t group)

  - Kernel-Datei: `fs/open.c`
  - Audit-Datei: `audit/audit_open.c`
- Audit-Event: `AUE_CHOWN(35), AUE_FCHOWN(36), AUE_LCHOWN(37)`
- variable Parameter
  - Datei 1: `filename`, bzw. der Dateiname hinter `fd`
  - Arg 1: `user`
  - Arg 2: `group`

sys_utimes(char * filename, struct timeval * utimes)

- Kernel-Datei: `fs/open.c`
- Audit-Datei: `audit/audit_open.c`

- Audit-Event: `AUE_UTIMES(38)`
- variable Parameter
  - Datei 1: `filename`
  - Arg 1: `times[0].tv_sec`, wenn `times == NULL` dann `CURRENT_TIME`
  - Arg 2: `times[1].tv_sec`, wenn `times == NULL` dann `CURRENT_TIME`

- Systemruf: `sys_utime(char *filename, struct utimbuf * times)`
  - Kernel-Datei: `fs/open.c`
  - Audit-Datei: `audit/audit_open.c`
- Audit-Event: `AUE_UTIME(39)`
- variable Parameter
  - Datei 1: `filename`
  - Arg 1: `times.actime`, wenn `times == NULL` dann `CURRENT_TIME`
  - Arg 2: `times.modtime`, wenn `times == NULL` dann `CURRENT_TIME`

sys_chroot(const char * filename)

- Kernel-Datei: `fs/open.c`
- Audit-Datei: `audit/audit_open.c`

- Audit-Event: `AUE_CHROOT(40)`
- variable Parameter
  - Datei 1: `filename`
- Besonderheiten: Das Erzeugen des Audit-Records ist auf die zwei Funktionen `audit_chroot_pre(...)` und `audit_chroot_post(...)` verteilt. Die pre-Funktion ermittelt, bevor das root-Verzeichnis des aktuellen Prozesses verändert wird, den absoluten Pfad von `filename`.

sys_truncate(const char *path, unsigned long length)
sys_ftruncate(unsigned int fd, unsigned long length)

- Kernel-Datei: `fs/open.c`
- Audit-Datei: `audit/audit_open.c`

- Audit-Event: `AUE_TRUNCATE(41), AUE_FTRUNCATE(42)`
- variable Parameter
  - Datei 1: `filename`, bzw der Dateiname hinter `fd`
  - Arg 1: `length`

sys_syslog(int type, char * buf, int len)

- Kernel-Datei: `kernel/printk.c`
- Audit-Datei: `audit/audit_printk.c`

- Audit-Event: `AUE_SYSLOG(43)`
- variable Parameter
  - Arg 1: `type`

sys_execve(struct pt_regs regs)

- Kernel-Datei: `arch/i386/kernel/process.c`
  `fs/exec.c`
- Audit-Datei: `audit/audit_process.c`

- Audit-Event: `AUE_EXECVE(45)`
- variable Parameter
  - Datei 1: `regs.ebx`, Name der auszuführenden Datei
  - Env: `regs.ecx` und `regs.edx`, das Programm-Environment

- Besonderheiten: Das Füllen des Audit-Records verteilt sich auf die beiden Funktionen `audit_execve_pre(...)` und `audit_execve_post(...)`. Da während der Ausführung von `sys_execve(...)` die Inhalte der Register überschrieben werden, müssen die darin enthaltenen Werte vor der Ausführung des Systemrufs gesichert werden, dies erfolgt in der Funktion `audit_execve_pre(...)`.

sys_ptrace(long request, long pid, long addr, long data)

- Kernel-Datei: `arch/i386/kernel/ptrace.c`
- Audit-Datei: `audit/audit_ptrace.c`
- Audit-Event: `AUE_VTRACE(46)`
- variable Parameter
  - Arg 1: `pid`
  - Arg 2: `request`
- Besonderheiten: Sollte die `pid` gleich der Prozess-ID des Audit-Dämons sein, wird der Systemruf nicht aufgezeigt, da sonst die Gefahr eines Deadlocks besteht.

sys_quotaclt(int cmd, const char *special, int id, caddr_t addr)

- Kernel-Datei: `fs/dquota.c`
  `fs/noquota.c`
- Audit-Datei: `audit/audit_quot.c`
- Audit-Event: `AUE_QUOTACTL(47)`
- variable Parameter
  - Datei 1: `special`
  - Arg 1: `cmd`
  - Arg 2: `id`
- Besonderheiten: Sollte die Disk-Quota Unterstützung nicht aktiviert sein, ist die Funktion `sys_quotactl(...)` in der Datei `fs/noquota.c` implementiert und gibt immer den Fehler `ENOSYS` zurück.

sys_accept(int fd,struct sockaddr *addr, int *addrlen)

- Kernel-Datei: `net/socket.c`
- Audit-Datei: `audit/audit_socket.c`
- Audit-Event: `AUE_ACCEPT(48)`
- variable Parameter
  - Datei 1: `AF_INET`, `AF_INET6` die IP-Adresse des Clients in Binärform,
    `AF_UNIX` der Pfadname
  - Arg 1: Adressfamilie (`AF_INET`, `AF_INET6` oder `AF_UNIX`)
  - Arg 2: `AF_INET`, `AF_INET6` Port-Nummer,
    `AF_UNIX` gleich `0`
- Besonderheiten: Es ist zu beachten, dass mit der Abspeicherung von IP-Adressen in Binärform das Record-Format beeinflusst werden kann. D.h. mitunter enthält der Record mehr als 5 Stringend-Kennungen. (siehe Abschnitt 4.2.2)

sys_connect(int fd,struct sockaddr *addr, int *addrlen)

- Kernel-Datei: `net/socket.c`
- Audit-Datei: `audit/audit_socket.c`
- Audit-Event: `AUE_CONNECT(49)`
- variable Parameter
  - Datei 1: `AF_INET`, `AF_INET6` die IP-Adresse des Servers in Binärform,
    `AF_UNIX` der Pfadname
  - Arg 1: Adressfamilie (`AF_INET`, `AF_INET6` oder `AF_UNIX`)
  - Arg 2: `AF_INET`, `AF_INET6` Port-Nummer,

AF_UNIX gleich 0

- Besonderheiten: siehe sys_accept(...)

sys_mount(char *dev_name, char *dir_name, char *type, unsigned long new_flags, void *data)

- Kernel-Datei: fs/super.c
- Audit-Datei: audit/audit_super.c
- Audit-Event: AUE_MOUNT(50)
- variable Parameter
  - Datei 1: dev_name
  - Datei 2: dir_name
  - Arg 1: new_flags
  - Arg 2: ein Hash-Wert für type
- Besonderheiten: Sollte der Typ gleich nfs sein, befindet sich in dev_name eine Angabe wie: nikita:/home/ost. Damit wird vom nfs-Server nikita das Verzeichnis /home/ost gemountet. Da dieses auf dem nfs-Client nicht genauer beschrieben werden kann, wird in dev_name lediglich die übergebene Zeichenkette gespeichert. Alle weiteren Angabe zu Datei 1 bleiben leer.

sys_swapon(const char *specialfile, int swap_flags)

- Kernel-Datei: mm/swapfile.c
- Audit-Datei: audit/audit_swapfile.c
- Audit-Event: AUE_SWAPON(51)
- variable Parameter
  - Datei 1: specialfile
  - Arg 1: swap_flags

sys_swapoff(const char *specialfile)

- Kernel-Datei: mm/swapfile.c
- Audit-Datei: audit/audit_swapfile.c
- Audit-Event: AUE_SWAPOFF(52)
- variable Parameter
  - Datei 1: specialfile

sys_setuid(uid_t uid)

- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETUID(53)
- variable Parameter
  - Arg 1: uid

sys_setgid(gid_t gid)

- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETGID(54)
- variable Parameter
  - Arg 1: gid

sys_setreuid(uid_t ruid, uid_t euid)

- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETREUID(55)
- variable Parameter

- Arg 1: ruid
- Arg 2: euid

sys_setregid(gid_t rgid, gid_t egid)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETREGID(56)
- variable Parameter
  - Arg 1: rgid
  - Arg 2: egid

sys_sethostname(char *name, int len)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETHOSTNAME(57)
- variable Parameter
  - Datei 1: name

sys_setdomainname(char *name, int len)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETDOMAINNAME(58)
- variable Parameter
  - Datei 1: name

sys_setgroups(int gidsetsize, gid_t *grouplist)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETGROUPS(59)
- variable Parameter
  - Arg 1: gidsetsize
  - Arg 2: grouplist[0]
- Besonderheiten: Sollte gidsetsize größer als 1 sein, gehen die zusätzlich übergebenen Gruppen verloren. Da im Record nur Platz für zwei long-Werte ist, können keine weiteren Gruppen gespeichert werden. Sollte eine Record auftreten, in dem gidsetsize größer 1 ist, kann der Administrator immer noch aus der /etc/groups die zusätzlichen Gruppen ermitteln.

sys_setfsuid(uid_t uid)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETFSUID(60)
- variable Parameter
  - Arg 1: uid

sys_setfsgid(gid_t gid)
- Kernel-Datei: kernel/sys.c
- Audit-Datei: audit/audit_sys.c
- Audit-Event: AUE_SETFSGID(61)
- variable Parameter
  - Arg 1: gid

sys_setresuid(uid_t ruid, uid_t euid, uid_t suid)

- Kernel-Datei: `kernel/sys.c`
- Audit-Datei: `audit/audit_sys.c`

- Audit-Event: `AUE_SETRESUID(62)`
- variable Parameter
  - Arg 1: `ruid`
  - Arg 2: `rgid`
- Besonderheiten: Aus Platzgründen kann die `suid` nicht gespeichert werden.

sys_setresgid(gid_t rgid, gid_t egid, gid_t sgid)

- Kernel-Datei: `kernel/sys.c`
- Audit-Datei: `audit/audit_sys.c`

- Audit-Event: `AUE_SETRESGID(63)`
- variable Parameter
  - Arg 1: `rgid`
  - Arg 2: `egid`
- Besonderheiten: siehe `sys_setresuid(...)`

sys_stime(int *tptr)

- Kernel-Datei: `kernel/time.c`
- Audit-Datei: `audit/audit_time.c`

- Audit-Event: `AUE_STIME(64)`
- variable Parameter
  - Arg 1: `tptr`

sys_settimeofday(struct timeval *tv, struct timezone *tz)

- Kernel-Datei: `kernel/time.c`
- Audit-Datei: `audit/audit_time.c`

- Audit-Event: `AUE_SETTIMEOFDAY(65)`
- variable Parameter
  - Arg 1: `tv.tv_sec`, wenn `tv == NULL` dann `0`
  - Arg 2: `tv.tv_usec`, wenn `tv == NULL` dann `0`

sys_adjtimex(struct timex *txc_p)

- Kernel-Datei: `kernel/time.c`
- Audit-Datei: `audit/audit_time.c`

- Audit-Event: `AUE_ADJTIME(66)`
- Besonderheiten: Die Struktur `timex` bietet zuviele Möglichkeiten die Systemzeit zu beeinflussen, dass eine Aufzeichnung in den durch das Record-Format vorgebenden Einträgen nicht sinnvoll realisierbar ist.

B.2 Zusätzliche Audit-Events

audit_sysboot()

Da das Audit-Modul beim Starten des Rechners noch nicht aktiviert sein kann, wird das boot-Event beim Starten des Audit-Dämons erzeugt. Der Boot-Zeitpunkt wird aus der aktuellen Systemzeit und der uptime errechnet.

Im Gegensatz zu allen anderen Audit-Events wird beim boot-Event der statische Teil des Audit-Records nicht entsprechend des Kontextes des aufrufenden Prozesses gefüllt. Der aufrufende Prozess ist der Audit-Dämon, das boot-Event erfolgt allerdings viel eher und nicht in dessen Kontext. Aus diesem Grund wird der statische Teil mit der Werten des init-Prozesses gefüllt und im Eintrag `arg1` der boot-Zeitpunkt vermerkt.

Das boot-Event hat die ID `AUE_SYSTEMBOOT(67)`.

audit_setaid()

Das Setzen der Audit-ID erfolgt in der Regel nur beim Starten des Systems und beim Anmelden eines neuen Nutzers. Sollte das Ändern der Audit-ID zu einem anderen Zeitpunkt auftreten, ist dies mitunter ein Indiz für eine Sicherheits- bzw. Policy-Verletzung des aufrufenden Prozess. Aus diesem Grund wird vom Audit-Modul das Event `AUE_SETAID` generiert.

Der statische Teil des Audit-Records wird entsprechend dem aufrufenden Prozesses gesetzt. Der variable Teil erhält die Einträge:

- Event: `AUE_SETAUID(70)`
- Arg 1: `aid` des Nutzers

B.3 Applikation-Events

Im Kapitel 4wurde bereits darauf hingewiesen, dass die Applikationen **`login`**, **`kdm`** und **`su`** hinsichtlich eines Applikation-Events erweitert wurden. In diesem Abschnitt erfolgt eine etwas genauere Erläuterung dieser Events.

B.3.1 login-Event

Da die Aktion Login oder der Versuch sich anzumelden schon ein relevantes Ereignis ist, wird bei jedem Ablauf des Programms `login` ein Audit-Event generiert und gespeichert. Dabei werden auch die fehlgeschlagenen Versuche protokolliert, da diese möglicherweise Aufschluss über einen möglichen Einbruchsversuch geben und später entsprechend ausgewertet werden müssen.

Im Audit-Record werden neben dem statischen Teil folgende Einträge vermerkt:

- Event: `AUE_LOGIN(100)`
- Arg 1: `uid` des Nutzers
- Arg 2: IP-Adresse des Herkunftsrechners

Da alle nachfolgenden Prozesse einem neuen Nutzer zugeordnet werden und damit der login-Prozess eine neue Audit-ID und eine neue Session-ID benötigt und diese am besten vom `login`-Programm eingebracht werden, war es ohne hin notwendig diese Programme zu erweitern.

Erfolgte der Zugang von einem entferntem Rechner aus, muss außerdem die IP-Adresse des Rechners, von dem aus der Zugriff erfolgt, aufgezeichnet werden. Um den Audit-System dies mitzuteilen, ruft das login-Programm nach der folgreichen Authentifizierung des Nutzers `audit_login(...)` auf. Daraufhin wird im Kernel die Audit-Struktur in der Task-Struktur des aktuellen Prozesses aktualisiert.

Login-Programm

Das Programm **`login`** dient als Zugangspunkt zum Linux-System. Dabei kann der Zugang lokal oder über das Netzwerk erfolgen. Der Nutzer identifiziert und authentifiziert sich mittels des `login`-Programms gegenüber dem System. War das Einloggen erfolgreich, bekommt der Nutzer ein Kontrollterminal zugeordnet und kann sich im Rahmen seine Policies im System bewegen.

KDM-Dämon

Beim KDM-Dämon handelt es sich um den vom KDE-Projekt erweiterten XDM-Dämon. Beide Programme dienen als grafisches Login. Die Funktionsweise ist ähnlich dem normalen login-Programm. Der Nutzer muss sich an dieser Stelle dem System gegenüber identifizieren und authentifizieren. War dies erfolgreich, wird für den Nutzer eine XSession gestartet.

Für den Zugang über das Netzwerk ist das XDMCP-Protokoll verantwortlich. Wurden der KDM-Dämon mit dieser Option kompiliert, kann der Zugang von einem entferntem Rechner aus erfolgen.

Der Mechanismus ist ähnlich dem lokalem Login, einzig die Ein- und Ausgaben wird über das Netzwerk zwischen den Rechnern übertragen.

Es werden die gleichen Daten wie beim `login`-Programm erfasst.

B.3.2 su-Event

Das Programm **`su`** (switch user) dient zum temporären Wechsel der Nutzeridentität. Hierzu wird im Kernel für den aktuellen Prozess die Nutzer und die Gruppen ID verändert. Auf einem UNIX-System ist **`su`** die schnellste Möglichkeit, Aktionen mit einer anderen Identität und damit unter Umständen mit anderen Privilegien auszuführen. Für eine spätere Identifikation des Nutzer wird die Audit-ID mitgeführt. Sie darf durch das `su`-Programm nicht verändert werden, da die unter der neuen Identifikation ausgeführten Aktionen immer noch dem ursprünglichem Nutzer zugeordnet werden müssen.

Im Audit-Record das `su`-Event wird im Eintrag `arg1` die `uid` und im Eintrag `arg2` die `gid` des gewünschten Nutzers vermerkt. Existiert der Nutzer nicht auf dem System, wird sowohl in `arg1` als auch in `arg2` der Wert `-1` eingetragen. Das su-Event hat die Event-ID `102` und wird mittels des Defines `AUE_SU` eingetragen.

Anhang C

Technische Beschreibung

C.1 Datenstrukturen

In diesem Abschnitt erfolgt eine kurze Beschreibung der im Rahmen von LOSA definierten Datenstrukturen. Die hier aufgeführten Strukturen sind aus dem neuesten Release vom LOSA entnommen und entsprechen aus diesem Grund nicht immer ganz den bisherigen Darstellungen und Beschreibungen. In den Erläuterungen zu den Strukturen werden die Unterschiede im einzelnen dargestellt und erklärt.

/usr/include/linux/acl.h

Die Datei `/usr/include/linux/acl.h` enthält die Kernel-internen Strukturen zur Verwaltung der ACL, dazu zählen die ACEs und die Elemente der REL.

```
struct acl_rel_entry {
      int                          type;
      int                          id;
      unsigned long                au_evt_used[AU_MASK_SIZE / 2];
      unsigned long                au_evt_mask[AU_MASK_SIZE];
      unsigned long                op_used;
      unsigned long                op_mask;
      struct acl_entry             *ace;
      struct acl_rel_entry         *prev;
      struct acl_rel_entry         *next;
};
```

Die Struktur `acl_rel_entry` dient der Verwaltung der REL. Im Gegensatz zur Darstellung im Abschnitt 4.1.3.1 sind die Einträge `au_evt_used`, `op_used`, `op_mask`, `ace` und `prev` hinzugekommen. Die Einträge `op_used` und `op_mask` dienen der Durchsetzung einer Zugriffskontrolle wie bei Windows NT. Bisher ist lediglich eine Implementierung für den Systemruf `execve(...)` realisiert. Die Maske selbst hat Platz für weitere 31 Systemruf. Die übrigen neuen Elemente dienen der Verwaltung der REL.

```
struct acl_entry {
      char                         *name;
      unsigned long                hash;
      struct acl_rel_entry         *rel;
      struct acl_entry             *sib_r;
      struct acl_entry             *cld;
      struct acl_entry             *sib_l;
      struct acl_entry             *parent;
};
```

Wie bei der Struktur `acl_rel_entry` sind in der Struktur der ACE einige neue Einträge für die Verwaltung hinzugekommen. Die Bedeutung aller übrigen Einträge entspricht der Beschreibung aus Abschnitt 4.1.3.

/usr/include/linux/audit.h

Die Datei `/usr/include/linux/audit.h` ist der eigentliche Header des Audit-Moduls. Hier sind fast alle Strukturen, die innerhalb des Moduls benötigt werden, definiert. Darüber hinaus enthält die Datei alle notwendigen Defines, die hier nicht erläutert werden.

```
struct audit_app_event {
      int                          status;
      unsigned long                arg1;
      unsigned long                arg2;
      char                         file[PATH_MAX + 1];
      char                         file2[PATH_MAX + 1];
      char                         *env;
};
```

Die Struktur `audit_app_event` wird dem Systemruf `audit_event(...)` übergeben und enthält die Nutzer-spezifischen Einträge eines Applikation-Events. Der Eintrag `env` kann eine beliebige mit einer Stringend-Kennung terminierte Zeichenkette sein. Sie wird im Audit-Record im Eintrag `env` hinterlegt und kann damit bis zu 128 kByte groß sein.

```
struct audit_prog_id {
      char                         comm[16];
      unsigned long                inode;
      kdev_t                       dev;
};
```

Die Struktur `audit_prog_id` wird für die Vergabe von Capabilities verwendet und enthält die Beschreibung eines Programms.

```
struct audit_cap_db {
      int                          aid;
      struct audit_prog_id         prog_id;
      int                          cap;
      struct audit_cap_db          *next;
};
```

In der Struktur `audit_cap_db` werden die Capabilities des Audit-Systems verwaltet. Handelt es sich bei dem Capability-Besitzer um einen Nutzer, ist dessen Audit-ID im Eintrag `aid` abgelegt. Bei einem Programm bspw. **`su`** wird die Struktur `audit_prog_id` gefüllt.

```
struct audit_exec_buffer {
      int                          pages;
      int                          state;
      char                         *addr;
      char                         *arg;
      char                         *env;
      unsigned int                 arg_len;
      unsigned int                 env_len;
      struct audit_exec_buffer     *next;
};
```

Wie im Abschnitt 4.1.5 beschrieben wird zur Verwaltung des Programm-Environments eine spezielle Struktur verwendet. Hierbei handelt es sich um die Struktur `audit_exec_buffer`. Eine genaue Beschreibung der einzelnen Elemente ist im Abschnitt 4.1.5.1 zu finden.

```
struct audit_user_exec_buffer {
      char                         data[ARG_MAX];
```

```
    unsigned int                arg_len;
    unsigned int                env_len;

};
```

Für die Struktur `audit_exec_buffer` existiert noch eine zweite Variante. Sie wird lediglich im Nutzermodus verwendet. Die Kernel-Variante enthält eine Vielzahl von Einträgen, die der Einsparung von Speicherplatz dienen. Im Nutzermodus wird für die Speicherung des Programm-Environments eine Struktur mit einem statischen Puffer `data` verwendet. Dieser hat immer eine Größe von 128 kByte. Zusätzlich werden lediglich die Längenangabe für `arg` und `env` in der Struktur vermerkt.

```
struct audit_record {
    time_t                      date;
    int                         aid;
    uid_t                       ruid;
    gid_t                       rgid;
    uid_t                       euid;
    gid_t                       egid;
    pid_t                       session;
    pid_t                       pid;
    unsigned long               tty;
    int                         status;
    int                         event;
    uid_t                       owner;
    gid_t                       gowner;
    unsigned long               dev;
    unsigned long               perm;
    unsigned long               inode;
    unsigned long               from;
    unsigned long               arg1;
    unsigned long               arg2;
    struct audit_exec_buffer    *env;
    char                        hostname[MAXHOSTNAMELEN + 4];
    char                        file[PATH_MAX + 4];
    char                        file2[PATH_MAX + 4];
    unsigned long               hostname_len;
    unsigned long               file_len;
    unsigned long               file2_len;
};
```

Die Struktur des Audit-Records `audit_record` ist ebenfalls in der Datei `linux/audit.h` definiert. Eine genaue Beschreibung der einzelnen Einträge erfolgte bereits ausführlich in den vorangegangenen Abschnitten.

```
struct audit_buffer {
    int                         state;
    unsigned long               type;
    pid_t                       pid;
    struct audit_record         *entry;
    struct audit_buffer         *next;
    struct audit_buffer         *prev;
};
```

Die Verwaltung der Audit-Records erfolgt mittels der Struktur `audit_buffer`. Im Abschnitt 4.1.5.1 wurden bereits die Einträge `state`, `entry`, `next` und `prev` erläutert. Der Eintrag `type` ist für eine Erweiterung des Audit-Moduls hinsichtlich verschiedener Audit-Record-Typen vorgesehen. Bisher

sind lediglich Records vom Typ `host` bekannt, doch bereits mit der Implementierung von HONA würde ein zweiter Record-Typ hinzukommen. Der Eintrag `pid` wird für die Zuordnung von Record und Prozess verwendet. Er wird beim Anfordern des Records gesetzt und dient, falls der anfordernde Prozess in einem überwachtem Systemruf terminiert, zum Aufräumen des Audit-Puffers. In diesem Fall könnte der Eintrag `state` noch auf dem Wert `AU_USED` steht und könnte nie zurückgesetzt werden. Eine garbage-Routine, aufgerufen durch die core-dump-Funktion des Kernels, durchsucht beim Terminieren des Prozess den Audit-Puffer nach Records des beendeten Prozesses. Wurde eine Record des Prozesses gefunden, der zudem nicht korrekt gefüllt wurde, wird dieser für `AU_UNUSED` erklärt.

/usr/include/linux/sched.h

Die Datei `/usr/include/linux/sched.h` gehört nicht zum Audit-Modul. Aus Programmier-technischen Gründen war es jedoch notwendig, die Audit-Struktur in dieser Datei zu definieren.

```
struct audit_struct {
      atomic_t                    count;
      int                         aid;
      gid_t                       agid;
      pid_t                       sid;
      unsigned long               from;
      int                         set;
};
```

Eine genaue Beschreibung der einzelnen Einträge ist im Abschnitt 4.1.2.1 zu finden.

/usr/include/acl.h

Zur permanenten Speicherung der ACL wird, wie im Abschnitt 3.2.3 erläutert, die Funktionalität der gdbm-Bibliothek genutzt. Die einzelnen Datensätze werden binär bzw. als Zeichenkette abgelegt. Die Strukturen der Datei `/usr/include/acl.h` definieren das Format der Datensätze.

```
struct evt_db_struct {
      unsigned long               evt_mask[AU_MASK_SIZE];
      unsigned long               evt_mod[AU_MASK_SIZE / 2];
};
```

Die Struktur `evt_db_struct` wird zur Speicherung der Preselectionsmaske verwendet. Die zugehörige ID ist im Schlüssel des gdbm-Datensatzes enthalten.

```
struct acl_struct {
      unsigned long               evt_mask[AU_MASK_SIZE];
      char                        path[PATH_MAX];
};
```

Die Struktur `acl_struct` wird zur Speicherung der ACEs verwendet. Wie bei der Preselectionsmaske ist die ID im Schlüssel enthalten.

/usr/include/audit_trail.h

Die Datei `/usr/include/audit_trail.h` enthält die notwendigen Strukturen zum Laden eines Audit-Records mittels der vorgegebenen Bibliotheksfunktionen. Jede Funktion erwartet mindestens einen Zeiger auf eine der hier beschriebenen Strukturen.

```
struct audit_trail_entry {
      unsigned long               no;
      unsigned long               pos;
      unsigned long               type;
```

```
    size_t                      data_size;
    void                        *data;
};
```

Da im Audit-Trail Records verschiedenen Typs und Formats enthalten sein können, muss jeder Record durch einen Header beschrieben werden. Jede Funktion zum Laden eines Records gibt einen Zeiger auf eine Struktur vom Typ `audit_trail_entry` zurück. Der eigentliche Record ist im Element `data` gespeichert. Die übrigen Einträge beschreiben dessen Typ und dessen Position in der Audit-Datei.

```
struct audit_tdb_entry {
    unsigned long               no;
    unsigned long               pos;
    unsigned long               type;
    struct audit_tdb_entry      *next;
    struct audit_tdb_entry      *prev;
};
```

Zur Steigerung der Performanz kann für eine Audit-Datei eine Datenbasis angelegt werden. Sie beschreibt den Aufbau der Audit-Datei. Für jeden Record wird in der Datenbasis ein Element vom Typ `audit_tdb_entry` angelegt. Es beschreibt die Position (`pos`) des Records in der Datei und dessen Typ (`type`). Der Typ wird beim Laden des Records benötigt, da davon das Format und die Größe des Records abhängen.

```
struct audit_trail_db {
    char                        trail_id[AU_TRAIL_ID_SIZE];
    unsigned long               recs;
    struct audit_tdb_entry      *first;
    struct audit_tdb_entry      *current;
};
```

Die Struktur `audit_trail_db` dient als Header für die Trail-Datenbasis. Die `trail_id` enthält die ersten `AU_TRAIL_ID_SIZE`-Bytes der Audit-Datei und dient zu dessen Identifizierung. Im Eintrag `recs` wird die Anzahl der in der datei gespeicherten Records abgelegt.

C.2 Bibliotheksfunktionen

Wie im Abschnitt 3.2.2 erläutert, erfolgt die Kommunikation mit dem Audit-Modul über einen einzigen Systemruf, welcher von einer Vielzahl von Wrapper-Funktionen mit den richtigen Parametern aufgerufen wird. Im folgenden werden die Wrapper-Funktionen kurz beschrieben, darüber hinaus enthält dieser Abschnitt die Beschreibung der Funktionen zum Einlesen des Audit-Trails.

/usr/include/audit.h

Die Datei `/usr/include/audit.h` enthält die administrativen Funktionen des Audit-Systems. Die hier vorgestellten Funktionen sind alle in der Bibliothek `libaudit.so` implementiert und können mittels `-laudit` beim Übersetzen eines Programms eingebunden werden.

`int auditcall __P ((int __call, unsigned long *args));`

- Parameter: `__call`: ist die Systemruf-ID der Systemfunktion `sys_auditcall()`
  `args`: ist eine Zeiger auf die Argumente des Systemrufs
- Beschreibung: Die Funktion `auditcall()` ist der allgemeine Zugangspunkt zum Audit-System. Sie sollte lediglich von Bibliotheksfunktionen verwendet werden.

`int audit_service(int fd, unsigned long limit);`

- Parameter: `fd`: ist der Dateideskriptor der aktuellen Audit-Datei

limit: ist die maximale Anzahl an Bytes, die in noch die Datei geschrieben werden dürfen

- Beschreibung: Die Funktion audit_service() speichert die Audit-Records des Kernel-Puffers in die durch fd spezifizierte Datei. Ist die Menge der zu speichernden Daten größer als limit, wird das Speichern beim ersten Überschreiten der Grenze abgebrochen und der Fehler E2BIG zurückgegeben.

int audit_ctl(int cmd, unsigned long *arg);

- Parameter: cmd: Kommando-ID der aufzurufenden Funktion
  arg: Zeiger auf die Adresse der Parameter der aufzurufenden Funktion
- Beschreibung: audit_ctl() ist der allgemeine Zugangspunkt zu den Managementfunktionen des Audit-Systems. Sie wird vorwiegend innerhalb der Bibliothek verwendet.

***int audit_event(unsigned int evt_id,
struct audit_app_event *evt_data);***

- Parameter: evt_id: ID des zu generierenden Events
  evt_data: Zeiger auf eine Audit-Event-Struktur, mit den zu protokollierenden Daten (siehe Anhang C.1).
- Beschreibung: Mittels der Funktion audit_event() kann ein Nutzer-spezifisches Audit-Event erzeugt werden. Ist der Parameter evt_data ein Null-Zeiger, wird ein Audit-Event mit der Event-ID evt_id und dem Status failed erzeugt. Für den Aufruf der Funktion wird die Capability AU_CAP_MSG benötigt.

int audit_on(int buf_size);

- Parameter: buf_size: Anzahl der Kernel-internen Audit-Records
- Beschreibung: Die Funktion audit_on() initialisiert den Kernel-internen Audit-Record-Puffer und aktiviert die Protokollierung.

int audit_off();

- Parameter: -
- Beschreibung: Die Funktion audit_off() deaktiviert die Protokollierung und löscht den Kernel-internen Audit-Record-Puffer.

int audit_inituser(int aid);

- Parameter: aid: Audit-ID (Nutzer-ID) des zu initialisierenden Nutzers
- Beschreibung: Die Funktion audit_inituser() initialisiert das Audit-Environment des Nutzers mit der Nutzer-ID aid. Dazu zählt das Aktualisieren der Preselektionsmasken aller Prozesse des Nutzer und das Einfügen der acl-Regeln für diesen Nutzer. Die Funktion wird beim Starten des Audit-Systems zum Initialisieren bereits angemeldeter Nutzer verwendet.

int audit_login(int aid, gid_t agid, pid_t sid, unsigned long from);

- Parameter: aid: Audit-ID des zu initialisierenden Nutzers,
  agid: Initiale Gruppen-ID des Nutzers,
  sid: Session-ID der login-Session (Prozess-ID des login-Prozesses)
  from: IP-Adresse (nur IPv4) des Herkunftsrechners
- Beschreibung: Ähnlich wie die Funktion audit_inituser() initialisiert die Funktion audit_login() das Audit-Environment des Nutzers mit der Nutzer-ID aid, jedoch wird lediglich der aktuelle Prozess aktualisiert. Zusätzlich wird der Inhalt der Audit-Struktur des aktuellen Prozesses auf die übergebenen Werte gesetzt. Die Funktion wird beim initialen Anmelden eines Nutzers aufgerufen, bspw. vom **login**-Programm.

int audit_setaid(int aid, int with_acl);

- Parameter: `aid`: Audit-ID für den aktuellen Prozess,
  `with_acl`: zusätzliche Initialisierung der Audit-Environmentes
- Beschreibung: Die Funktion `audit_setaid()` setzt die Audit-ID des aktuellen Prozesses auf `aid`. Ist der Parameter `with_acl` ungleich Null, wird zusätzlich das Audit-Environment des Nutzers initialisiert. Der Aufruf der Funktion ist nur erfolgreich, wenn die Audit-ID des aktuellen Prozesse aktualisiert werden darf.

int audit_setmask(int type, int id, int evt, int val);

- Parameter: `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `evt`: Identifikation des Events
  `val`: Wert des Events
- Beschreibung: Die Funktion `audit_setmask()` setzt den Wert des Events `evt` des durch die Parameter `type` und `id` beschriebenen Prozesses auf `val`. `id` kann sich auf nur einen Prozess, auf die Prozesse eines Nutzers, die Prozesse einer Gruppe von Nutzer oder alle Prozesse beziehen.

int audit_initmask(int type, int id, unsigned long *evt);

- Parameter: `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `evt`: Zeiger auf eine Event-Maske
- Beschreibung: Die Funktion `audit_initmask()` setzt die gesamte Event-Maske, der durch `type` und `id` identifizierten Prozesse, auf `evt`.

int audit_rb_reset(int size, int fd, unsigned long limit);

- Parameter: `size`: neue Größe des Kernel-internen Audit-Record-Puffers,
  `fd`: ist der Dateideskriptor der aktuellen Audit-Datei
  `limit`: ist die maximale Anzahl an Bytes, die in noch die Datei geschrieben werden dürfen.
- Beschreibung: Mittels der Funktion `audit_rb_reset()` kann die Größe des Kernel-internen Audit-Record-Puffers verändert werden. Die aktuell im Puffer enthaltenen Audit-Records werden in die durch `fd` spezifizierten Datei geschrieben.

int audit_eb_reset(int size, int order, int fd,
unsigned long limit);

- Parameter: `size`: neue Größe des `execve`-Prozess-Environment-Puffers,
  `order`: neue Page-Order für die Puffer-Segmente des Environment-Puffers,
  `fd`: ist der Dateideskriptor der aktuellen Audit-Datei,
  `limit`: ist maximale Anzahl an Bytes, die in noch die Datei geschrieben werden dürfen.
- Beschreibung: Mittels der Funktion `audit_eb_reset()` können die Parameter des `execve`-Prozess-Environment-Puffers verändert werden. Die aktuell im Kernel-internen Audit-Record-Puffer enthaltenen Audit-Daten werden in die durch `fd` spezifizierten Datei geschrieben.

int audit_eb_on(int size, int order);

- Parameter: `size`: Größe des `execve`-Prozess-Environment-Puffers,
  `order`: Page-Order der Puffer-Segmente des Environment-Puffers.
- Beschreibung: Die Funktion `audit_eb_on()` aktiviert den Kernel-internen `execve`-Prozess-Environment-Puffer.

int audit_eb_off(int fd, unsigned long limit);

- Parameter: `fd`: ist der Dateideskriptor der aktuellen Audit-Datei,
  `limit`: ist maximale Anzahl an Bytes, die in noch die Datei geschrieben werden dürfen.
- Beschreibung: Die Funktion `audit_eb_off()` deaktiviert den Kernel-internen `execve`-Prozess-Environment-Puffers. Die aktuell im Kernel-internen Audit-Record-Puffer enthaltenen Audit-Daten werden in den durch `fd` spezifizierten Audit-Trail geschrieben.

int audit_getmask(int type, int id, unsigned long *buf);
- Parameter: `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `buf`: Zeiger auf den Zielpuffer
- Beschreibung: Die Funktion `audit_getmask()` kopiert die Event-Maske des durch `type` und `id` spezifizierten Prozesses nach `buf`. Die Genauigkeit der zurückgegebenen Maske ist abhängig von der Granularität der Prozessauswahl mittels `type` und `id`. Es wird immer die Maske des ersten Prozesses, der die Kriterien von `type` und `id` erfüllt, zurückgegeben.

int audit_chmod(int mode);
- Parameter: `mode`: neuer Audit-Modus
- Beschreibung: Mittels der Funktion `audit_chmod()` kann der Betriebsmodus des Audit-Systems verändert werden. Befindet sich das System im **`secure`**-Modus gibt die Funktion immer den Fehler `EPERM` zurück.

int audit_getmod();
- Parameter: -
- Beschreibung: Der aktuelle Betriebsmodus des Audit-Systems kann mittels der Funktion `audit_getmod()` erfragt werden.

int audit_getadmin(int *aid);
- Parameter: `aid`: Zeiger auf den Rückgabewert
- Beschreibung: Die Funktion `audit_getadmin()` schreibt die Audit-ID des aktuell gültigen Audit-Administrators nach `aid`. Da die Audit-ID sowohl positiv als auch negativ sein kann, ist die Rückgabe über den Ergebniswert der Funktion nicht möglich.

int audit_setcap(int aid, int cap);
- Parameter: `aid`: Audit-ID
  `cap`: zusetzende Capability
- Beschreibung: Die Funktion `audit_setcap()` gibt der Audit-ID `aid` die Capability `cap`. **Achtung**: momentan kann die Capability nicht zurückgenommen werden.

int audit_setcap_prog(char *prog, int cap);
- Parameter: `prog`: Programmname mit absolutem Pfad
  `cap`: zusetzende Capability
- Beschreibung: Die Funktion `audit_setcap_prog()` gibt dem Programm `prog` die Capability `cap`. Das Programm wird bei der Überprüfung der Capability über den Name, die Inode und die Gerätenummer identifiziert, d.h. wird das Programm zu einem späteren Zeitpunkt verschoben, erlischt die Capability.

int audit_getcap(int aid, int *cap);
- Parameter: `aid`: Audit-ID

`cap`: Zeiger auf ein `int`-Array mit mindestens drei Elementen

- Beschreibung: Die Funktion `audit_getcap()` kopiert die Capabilities des Nutzers `aid` in das mittels `cap` übergebene Array. Jedes Element des Arrays steht für eine Capability, ist das Element ungleich Null, besitzt die Audit-ID diese Capability.

/usr/include/acl.h

Die Datei `/usr/include/acl.h` enthält die Funktionen zur Verwaltung der ACL. Da die ACL im aktuellen Release unabhängig vom Audit-System ist, werden die Bibliotheksfunktionen in einem separatem Header verwaltet. Die Bibliothek `libaudit.so` kann sowohl die Funktionen des Audit-Systems als auch die Funktionen zur Verwaltung der ACL enthalten, je nachdem, mit welchen Optionen die Bibliothek erzeugt wurde.

`int audit_setacl(char *res, int type, int id, int evt, int val);`

- Parameter: `res`: Ressourcenname mit absolutem Pfad,
  `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `evt`: Identifikation des Events
  `val`: Wert des Events
- Beschreibung: Die Funktion `audit_setacl()` setzt die Event-Maske des durch `res` spezifizierten ACEs. Die REL-Eintrag des ACE wird, wie bei der Funktion `audit_setmask()`, über die Parameter `type` und `id` spezifiziert.

`int audit_initacl(char *res, int type, int id, unsigned long *evt);`

- Parameter: `res`: Ressourcenname mit absolutem Pfad,
  `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `evt`: Zeiger auf eine Event-Maske
- Beschreibung: Die Funktion `audit_initacl()` initialisiert die Event-Maske des durch die Parameter `res`, `type` und `id` identifizierten REL-Eintrages mit der übergebenen Event-Maske `evt`.

`int acl_user_aumsk(int uid, int gid);`

- Parameter: `uid`: Nutzer-ID
  `gid`: Gruppen-ID des Nutzers `uid`
- Beschreibung: Die Funktion `acl_user_aumsk()` initialisiert die ACL des Nutzers `uid` aus der Gruppe `gid`. Hierzu werden mittels der gdbm-Bibliothek die notwendigen Daten aus der Datenbank ausgelesen und in den Kernel übertragen.

`int acl_setexec(char *res, int id, int type, int value);`

- Parameter: `res`: Ressourcenname mit absolutem Pfad
  `type`: Spezifikation des Parameters `id`,
  `id`: Identifikation des Prozesses, die Bedeutung der ID ist abhängig von `type`,
  `value`: 0 oder 1
- Beschreibung: Mittels der Funktion `acl_setexec()` ist es möglich, die exec-Permission des Dateisystems feingranularer zu vergeben. Die Parameter `id` und `type` spezifizieren die Prozesse, welchen die Ausführung des Programms, welches durch `res` beschrieben wird, untersagt bzw. gestattet wird.[13]

/usr/include/audit_trail.h

Um den Zugriff auf die Daten der Audit-Dateien zu erleichtern, werden in der Audit-Bibliothek die grundlegenden Funktionen bereits implementiert. Damit können die Anwenderprogramme unabhängig

[13] Die Funktion `acl_setexec()` ist nicht direkt Bestandteil von LOSA und wird aus diesem Grund in diesem Dokument nicht genauer erläutert.

vom genauen Format des Audit-Trails gestaltet werden Bei der Verwendung der im folgenden vorgestellten Funktionen müssen lediglich die notwendigen Strukturen bereitgestellt werden. Die Audit-Bibliothek beinhaltet I/O-Funktionen zum Füllen der Strukturen, dabei kann zwischen stream-orientierten und normalen Dateioperationen unterschieden werden. Für jede I/O-Funktion existiert eine stream-orientierten und einen nicht-stream-orientierte Implementierung. Wie bei den I/O-Systemfunktion beginnen die Stream-Funktionen mit einem kleinen f und erwarten einen Zeiger auf eine stream-Struktur anstelle eines Dateideskriptors.

Um den Zugriff auf die einzelnen Records der Audit-Dateien zu beschleunigen, kann eine Trail-Datenbasis auf gebaut werden. Diese enthält Informationen über die Anzahl der Records ib der Audit-Datei und deren genaue Position. Wird die Datenbasis nicht erstellt, muss unter Umständen bei jeder Leseoperation die gesamte Datei durchsucht werden.

int audit_init_tdb(int td, struct audit_trail_db *tdb);
int faudit_init_tdb(FILE *stream, struct audit_trail_db *tdb);

- Parameter: `td`, `stream`: aktuelle Audit-Datei,
  `tdb`: Zeiger auf eine Trail-Datenbasis
- Beschreibung: Die Funktion `audit_init_tdb()` erstellt eine Trail-Datenbasis für die mittels `td` bzw. `stream` übergebene Audit-Datei. Die eingelesenen Daten werden in der mittels `tdb` übergebenen Struktur gespeichert.

int audit_clear_tdb(struct audit_trail_db *tdb);

- Parameter: `tdb`: Zeiger auf eine Trail-Datenbasis.
- Beschreibung: Die Funktion `audit_clear_tdb()` gibt den für die Trail-Datenbasis angeforderten Speicher frei. Es wird lediglich der Speicher der Elemente der übergebenen Struktur freigegeben, die Struktur selbst bleibt gültig.

int audit_get_rec(int td, struct audit_trail_entry *entry);
int faudit_get_rec(FILE *stream, struct audit_trail_entry *entry);

- Parameter: `td`, `stream`: aktuelle Audit-Datei,
  `entry`: Zeiger auf eine Trail-Entry-Struktur
- Beschreibung: Die Funktion `audit_get_rec()` kopiert den aktuellen Record aus der Audit-Datei in die Struktur `entry`. Nach dem Lesen des Records ist der Dateizeiger einen Record weitergerückt.

```
int audit_get_rec_no(int td, unsigned long no,
                     struct audit_trail_entry *entry,
                     struct audit_trail_db *tdb)
int faudit_get_rec_no(FILE *stream, unsigned long no,
                      struct audit_trail_entry *entry,
                      struct audit_trail_db *tdb);
```

- Parameter: `td`, `stream`: aktuelle Audit-Datei
  `no`: Nummer des Records
  `entry`: Struktur zum Speichern des Records
  `tdb`: Zeiger auf die Trail-Datenbasis, kann `NULL` sein
- Beschreibung: Speichert den `no`-ten Record der Datei `td` bzw. `stream` in die Struktur `entry`. Wird dieser Funktion mittels `tdb` keine Trail-Datenbasis übergeben, muss unter Umständen die gesamte Datei durchsucht werden.

```
int audit_get_next_rec(int td, struct audit_trail_entry *entry,
                       struct audit_trail_db *tdb);
int faudit_get_next_rec(FILE *stream,
                        struct audit_trail_entry *entry,
                        struct audit_trail_db *tdb);
```

- Parameter: `td`, `stream`: aktuelle Audit-Datei,
  `entry`: Struktur zum Speichern des Records,
  `tdb`: Zeiger auf die Trail-Datenbasis
- Beschreibung: Die Funktion `audit_get_next_rec()` kopiert den Nachfolger des aktuellen Audit-Records aus dem Trail in die mittels `entry` übergebenen Trail-Entry-Struktur. Die Funktion erwartet zwingend einen Zeiger auf eine gültige Trail-Datenbasis.

```
int audit_get_prev_rec(int td, struct audit_trail_entry *entry,
                       struct audit_trail_db *tdb);
int faudit_get_prev_rec(FILE *stream,
                        struct audit_trail_entry *entry,
                        struct audit_trail_db *tdb);
```

- Parameter: `td`, `stream`: aktuelle Audit-Datei,
  `entry`: Struktur zum Speichern des Records,
  `tdb`: Zeiger auf die Trail-Datenbasis
- Beschreibung: Die Funktion `audit_get_prev_rec()` kopiert den Vorgänger des aktuellen Audit-Records aus dem Trail in die mittels `entry` übergebenen Trail-Entry-Struktur. Die Funktion erwartet zwingend einen Zeiger auf eine gültige Trail-Datenbasis.

```
int audit_retry_get_rec(int td, struct audit_trail_entry *entry);
int faudit_retry_get_rec(FILE *stream,
                         struct audit_trail_entry *entry);
```

- Parameter: `td`, `stream`: aktueller Audit-Trail,
  `entry`: Struktur zum Speichern des Records.
- Beschreibung: Sollte beim Einlesen des Trails ein Fehler aufgetreten sein, kann mittels der Funktion `audit_retry_get_rec()` versucht werden, einen neuen gültigen Record innerhalb des Trails zufinden. Die Funktion liest die nächsten 2 MB des aktuellen Trails ein und versucht darin einen gültigen Record zu finden. War dies erfolgreich, werden dessen Daten in die Trail-Entry-Struktur `entry` kopiert.

```
int audit_free_entry(struct audit_trail_entry *entry);
```

- Parameter: `entry`: Zeiger auf eine Trail-Entry-Struktur.
- Beschreibung: Die Funktion `audit_free_entry()` gibt den Speicher der Trail-Entry-Struktur frei.

C.3 Grammatiken der Shell- und Konfigurierungskommandos

Die Kommandos für das Konfigurationsinterface und den Audit-Dämon lassen sich mit einer Grammatik beschreiben. Sie wird in diesem Abschnitt vorgestellt. Die Bedeutung der einzelnen Schlüsselwörter wurde bereits in den vorangegangenen Abschnitten beschrieben und wird hier nicht noch einmal erläutert.

Die Grammatik für die Shell-Kommandos und für die Konfigurationsdatei des Audit-Dämon ist identisch. Der Parser des Konfigurationsinterfaces bzw. des Audit-Dämon behandelt eine Shell-Kommando wie eine Zeile in der Konfigurationsdatei. Kommandos, die an den Audit-Dämon geschickt werden, müssen mit dem Schlüsselwort *`auditd`* eingeleitet werden. Dem Schlüsselwort muss ein reguläres Kommando für den Audit-Dämon folgen. Diese werden nicht vom Konfigurationsinterface, sondern vom Audit-Dämon ausgewertet, so dass die Kommandos dessen Grammatik genügen müssen.

Konfigurationsinterface

Zunächst erfolgt die Auflistung der Grammatik des Konfigurationsinterfaces. Befehle, die an den Audit-Dämon weitergeleitet werden, werden im anschließenden Abschnitt beschrieben.

```
CMD ::=   "reset" | "terminate" | "next" | "flush" | "getout"
        | "noout" | "quit" | "getmod" | "getadmin"
        | "auditd " DAEMON
        | HELP | LOGIN | CHMOD | SETCAP | GETCAP | SETMASK
        | GETMASK | SETACL

HELP ::= "help" | "help auditd"

LOGIN ::= "login " ID

CHMOD ::= "chmod " MOD

SETCAP ::=   "aid : " ID " = " CAP
           | "prog : " PATH " = " CAP

GETCAP ::= "getcap " ID

SETMASK ::= "setmask " TARGET " " EVENTLIST

GETMASK ::= "getmask " TARGET

SETACL ::= "setacl " TARGET " " ACLLIST

AID ::= INT

MOD ::= "disabled" | "normal" | "secure"

CAP ::= "setaid" | "admin" | "msg"

PATH ::=   "/" | "/" DIRNAME "/" DIRNAME
         | "/" DIRNAME "/" | "/" DIRNAME

TARGET ::= "any" | "pid : " ID | "aid : " ID | "gid : " ID

EVENTLIST ::=   EVENTLIST " ; " EVENT
              | EVENT

EVENT ::=   "all : " EVT
          | "class : " CLASS " : " EVT
          | "id : " EID " : " EVT

EVT ::= "0" | ... | "3"

CLASS ::= "0" | ... | "31"

EID ::= "0" | ... | "127"

ACLLIST ::=   ACLLIST " ; " ACL
            | ACL

ACL ::=   "all : " EVT " : " PATH
        | "class : " CLASS " : " EVT " : " PATH
        | "id : " EID " : " EVT " : " PATH
```

```
ID ::= "'MIN_INT'" | ... | "'MAX_INT'"

DIRNAME ::= | ZEICHEN DIRNAME
            ZEICHEN

ZEICHEN ::=   "a" | ... | "z" | "A" | ... | "Z"
            | "0" | ... | "9" | "_" | "." | "-"
```

Audit-Dämon

Dieser Abschnitt enthält die Grammatik des Audit-Dämon. Meta-Worte, die in diesem Abschnitt nicht erläutert werden, genügen der Grammatik des Konfigurationsinterfaces.

```
DAEMON ::=   "state " STATE_CMD | "trail = " TRAIL
           | "limit = " LIMIT | "shutdown = " SHUTDOWN
           | "connect = " PATH | "pipe = " PATH
           | "warn = " WARN | "buffer : " BUFFER
           | "configfile = " PATH | "flush"

STATE_CMD ::=  "trail" | "limit" | "buffer" | "shutdown"
             | "warn" | "connect" | "configfile"

TRAIL ::=   PATH " : %" ID
          | PATH " : " LIMIT

LIMIT ::= "M" ID | "k" ID | ID

SHUTDOWN ::=   "script = " PATH
             | "audit"

WARN ::=   "script = " PATH
         | "no"

BUFFER ::=   " record : size = " SIZE
           | EXEC

EXEC ::=   "exec size = " EXEC_SIZE " order = " ORDER
         | "no"

SIZE ::= "'AU_BUFMIN'" | ... | "'INT_MAX'"

EXEC_SIZE ::= "0" | ... | "'INT_MAX'"

ORDER ::= "0" | ... | "5"
```

Audit-Ereignis-Konfigurationsoptimierer

Die Grammatik der Konfigurationsdatei des Audit-Ereignis-Konfigurationsoptimierers unterscheidet sich nur unwesentlich von der Grammatik des Konfigurationsinterfaces. Im Abschnitt D.5 wird eine Konfigurationsdatei vorgestellt. Die wesentlichen Unterschiede sollten in diesem Abschnitt deutlich werden.

C.4 Dateien von LOSA

In diesem Abschnitt erfolgt eine Aufstellung aller Quelltextdateien, man-Pages und Konfigurationsdateien eines LOSA-Paketes.

```
./:
COPYING                 INSTALL                       README
config                  install_redhat.sh
```

C.4.1 Konfigurationsdateien

```
./config/:
audit.conf              audit_acl.conf                audit_aid.conf
audit_evt.conf          auditd.conf
```

C.4.2 Audit-Bibliothek

```
./libaudit/:
ChangeLog               INSTALL                       Makefiles
Makefile.global         README

./libaudit/asm/:
Makefile                __auditcall.S

./libaudit/build/:
Makefile

./libaudit/csrc/:
Makefile                acl.c                         audit_ctl.c
audit_event.c           audit_service.c               audit_trail.c
audit_ver.c             audit_ver.h                   auditcall.c

./libaudit/include/:
acl.h                   audit.h                       audit_trail.h
auditcall.h             gnu-stabs.h                   syscall.h
sysdep.h                sysdep.i386.h
```

C.4.3 Man-Pages

```
./man/:
install_man.sh

./libaudit/man/man2/:
audit_chmod.2           audit_ctl.2                   audit_eb_off.2
audit_eb_on.2           audit_eb_reset.2              audit_event.2
audit_getadmin.2        audit_getcap.2                audit_getmask.2
audit_getmod.2          audit_initacl.2               audit_initmask.2
audit_inituser.2        audit_login.2                 audit_off.2
audit_on.2              audit_rb_reset.2              audit_service.2
audit_setacl.2          audit_setaid.2                audit_setcap.2
audit_setcap_prog.2     audit_setmask.2               auditcall.2

./man/man5/:
audit.conf.5            audit_acl.conf.5              audit_aid.conf.5
audit_evt.conf.5        auditd.conf.5

./man/man8/:
aca.8                   aclctl.8                      apwr.8
auditd.8
```

C.4.4 Patches

```
./patches/:
sh-utils-2.0-losa.patch util-linux-2.9w-losa.patch    Readme
Install                 kdebase-1.1.1-losa.patch

./patches/linux-2.4.0-test1/:
kernel_patch.sh         linux-2.4.0-test1-losa.patch
```

```
./patches/linux-2.4.0-test1/audit/:
Makefile                 acl.c                    audit_acct.c
audit_buffer.c           audit_cap.c              audit_capability.c
audit_ctl.c              audit_evt.c              audit_func.c
audit_ioport.c           audit_module.c           audit_namei.c
audit_open.c             audit_printk.c           audit_process.c
audit_ptrace.c           audit_quot.c             audit_socket.c
audit_super.c            audit_swapfile.c         audit_sys.c
audit_syscalls.c         audit_time.c

./patches/linux-2.4.0-test1/include/linux/:
acl.h                    audit.h                  audit_event.h
```

C.4.5 Script-Dateien

```
./scripts/:
audit                    audit_shutdown           audit_warn
awinit
```

C.4.6 Utilities

```
./utils/:
INSTALL                  Makefile                 Makefile.global
README
```

Audit-Konfigurationsinterface

```
./utils/aca/:
ChangeLog                Makefile                 cmd.c
cmd.h                    com.c                    com.h
help.c                   help.h                   lexprog.l
main.c                   yaccprog.y
```

Audit-Ereignis-Konfigurationsoptimierer

```
./utils/aclctl/:
Makefile                 db_structs.h             lexprog.l
main.c                   yaccprog.y
```

Audit-Wrapper

```
./utils/apwr/:
Makefile                 main.c
```

Trail-Viewer

```
./utils/atv/:
ChangeLog                Makefile                 atv
atv.devel                csrc                     tclsrc

./utils/atv/csrc/:
Makefile                 Makefile.include         atv.bin
main.c                   main.h                   new_commands.c
new_commands.h           search.c                 search.h
sprint_fperms.c

./utils/atv/tclsrc/:
audit_events.tcl         button_bar.tcl           file_dlg.tcl
id_list.tcl              info_page.tcl            list_wnd.tcl
menu_bar.tcl             messagebox.tcl           search_dlg.tcl
tv-main.tcl              version.tcl

./utils/atv/tclsrc/icons/:
bpoint.xpm               end.xpm                  exit.xpm
```

gpoint.xpm	load.xpm	mark.xpm
mark_all.xpm	next.xpm	prev.xpm
reload.xpm	top.xpm	unmark.xpm
unmark_all.xpm		

Audit-Dämon

./utils/auditd/:

ChangeLog	Makefile	com.c
com.h	core.c	core.h
help_stat.c	help_stat.h	lexprog.l
main.c	output.c	output.h
structs.h	yaccprog.y	

C.5 Geänderte Kernel-Quellen

Wie bereits erwähnt, war es an verschiedenen Stellen notwendig, den Originalquelltext des Linux-Kernel zu verändern. Dieser Abschnitt enthält alle geänderten Dateien.

```
./include/asm/unistd.h

./include/linux/sched.h
./include/linux/version.h

./arch/i386/kernel/entry.S
./arch/i386/kernel/init_task.c
./arch/i386/kernel/process.c
./arch/i386/kernel/ioport.c
./arch/i386/kernel/ptrace.c
./arch/i386/kernel/traps.c

./arch/i386/config.in

./kernel/fork.c
./kernel/sys.c
./kernel/capability.c
./kernel/module.c
./kernel/exit.c
./kernel/time.c
./kernel/printk.c
./kernel/acct.c

./fs/open.c
./fs/namei.c
./fs/exec.c
./fs/super.c
./fs/dquot.c
./fs/noquot.c

./Makefile

./mm/swapfile.c

./net/socket.c
```

Anhang D
Beispielkonfiguration

In diesem Abschnitt erfolgt die Beschreibung aller für das Audit-Modul notwendigen Konfigurationsdateien. Bei dem im folgenden abgedruckten Beispiel handelt es sich um die mit dem LOSA-Paketes mitgelieferten Konfiguration.

D.1 **Globale Konfiguration:** `audit.conf`

Die Datei `audit.conf` enthält die Konfiguration des init-Skriptes. Da das init-Skript mittels **`awk`** eingelesen wird und die verwendeten Regeln zum Parsen der Datei recht einfach gehalten sind, muss sich einem Kommentarzeichen `'#'` immer ein Leerzeichnen oder ein Tab anschließen. Ebenso müssen alle Schlüsselwörter, Zuweisungszeichen und Werte mit einem Leerzeichen oder Tab getrennt werden.

```
# -------------------------------------------------------------------------
#
#                 audit startup configuration file
#
#    Read the audit.conf(5) manual page in order to understand the options
#    listed here.
#
# -------------------------------------------------------------------------

# boot action (disabled, config, run)

boot = config

# runlevel (disabled, normal, secure)

mod = normal

# run application wrapper (manual, script)

app_wr = script

# audit capabilities

cap =       aid         101   admin \
            aid         101   msg \
            aid         101   setaid \

            # inetd
            aid         -10   setaid \
            aid         -10   msg \

            # mingetty
            aid         -20   setaid \
            aid         -20   msg \

            # program capabilities
```

```
            prog  /bin/su     msg
```

Bei der hier vorgegebenen Konfiguration wird das Audit-Modul beim Aufruf des init-Skripte lediglich konfiguriert. Der Audit-Dämon wird noch nicht gestartet. Damit dies geschieht, muss der Parameter `boot` auf dem Wert `run` gesetzt werden.

Die Festlegung des Betriebsmodus erfolgt mittels der Variable `mod`. Sollte der Modus auf `secure` gesetzt sein, steht dieser nach dem Hochfahren des Systems fest und kann nicht mehr geändert werden.

Der dritte Parameter `app_wr` legt fest, ob das Skript `awinit` zum Erzeugen der Wrapper-Skripte aufgerufen werden soll oder nicht. Dies kann bei jedem Neustart durch den Wert `script` oder wann immer notwendig per Hand erfolgen.

Abschließend erfolgt die Vergabe der Capabilities. Der Nutzer mit der `uid 101` ist in diesem Fall der Audit-Administrator und bekommt alle Capabilities zugewiesen. Der Nutzer `root` hat innerhalb des Audit-Moduls keine besonderen Privilegien, diese müssen ihm, falls dies gewünscht wird, explizit erteilt werden. Anschließend werden noch dem `inet`-Dämon, dem `mingetty` und dem Programm `su` die notwendigen Capabilities zugeteilt. Da der `inet`-Dämon und das `getty`-Programm mit einer festen Programm-ID und damit mit einer festen Audit-ID laufen, kann bei diesem Programmen die Capability an die Audit-ID gebunden werden. Das Programm `su` erhält bei jedem Aufruf eine andere Programm-ID und läuft zudem mit der Audit-ID des aufrufenden Nutzers. Da man lediglich dem Programm `su` eine Capability zusprechen möchte, muss dies über das Programm selbst erfolgen.

D.2 Konfiguration des Audit-Dämon: `auditd.conf`

Die Datei `auditd.conf` enthält die Konfiguration des Audit-Dämon. Das hier abgedruckte sollte im wesentlichen mit der späteren Konfiguration übereinstimmen. Mitunter müssen noch zusätzliche Trail-Verzeichnisse angegeben werden.

```
# ---------------------------------------------------------------------------
#
#                audit daemon startup configuration file
#
#    Read the auditd.conf(5) manual page in order to understand the options
#    listed here.
#
# ---------------------------------------------------------------------------

# directories for trail with ressourcen limits

#trail=/tmp : %10
trail=/tmp/foo : M5
trail=/var/audit/ : M5

# maximum trail size < 500 MB

limit=M20

connect=/var/run/auditd.pid

# first pipe for interactive modus

pipe=/tmp/pipe.1

# specify the initial buffer size

buffer : record : size=500
```

```
buffer : exec : size=10 order=0
# buffer : exec : no

# what should happen, if all ressources are emtpy

shutdown : script=/etc/security/audit_shutdown
#shutdown : audit

# audit warn scrpit

warn : script=/etc/security/audit_warn
#warn : no
```

Die einzelnen Parameter wurden bereits im Abschnitt 3.1.4 erläutert. Wichtig ist, dass der Audit-Administrator alle notwendigen Rechte für die hier verwendeten Verzeichnisse besitzt.

D.3 Vergabe von Audit-IDs: `audit_aid.conf`

Für eine minimale Verfolgung von Aktionen eines Dämon-Prozesses ist es notwendig, jedem Dämon eine Audit-ID zu geben, dies erfolgt mittels der Konfigurationsdatei `audit_aid.conf`. Jede Konfiguration setzt sich aus mindestens zwei Teilen zusammen. Der Angabe des absoluten Pfades zum Start-Script des Dämon bzw. zum Programm und der Zuordnung der Audit-ID zum Namen des Programms.

```
# ---------------------------------------------------------------------------
#
#                 audit ID configuration file
#
#    Read the audit_aid.conf(5) manual page in order to understand the
#    options listed here.
#
# ---------------------------------------------------------------------------

# path for the following executeables
path = /etc/rc.d/init.d

inet        -10
nfs         -11
portmap     -12
sendmail    -13
crond       -14
apmd        -15
syslog      -16
lpd         -17

path = /sbin

mingetty    -20 leader
getty       -21 leader
```

In der obigen Datei befinden sich die Dämon-Programme in zwei verschiedenen Verzeichnissen. Die ist zum einen `/etc/rc.d/init.d` und zum anderen `/sbin`. Nach der Angabe des Verzeichnisse erfolgt immer die Auflistung der Programme aus diesem Verzeichnis. Ein Verzeichnis bleibt solange gültig, bis ein neuer Pfad definiert wird.

Die Bedeutung des Schlüsselwortes `leader` erfolgte im Abschnitt 3.1.4.

D.4 Einteilung der Audit-Events in Gruppen: `audit_evt.conf`

Die Datei `audit_evt.conf` enthält die Konfiguration der Audit-Events. Sie ist unabhängig von der Kernel-Konfiguration und lediglich für eine einfachere Angabe der Event-IDs bei der Verwendung des Konfigurationsinterfaces notwendig.

Im ersten Abschnitt werden die vom Nutzer definierten Audit-Events aufgeführt. Sie haben immer ein Audit-ID größer als 100. Die Audit-Events 0 bis 99 sind für Kernel-Events reserviert.

Anschließend erfolgt die Einteilung der Events in Audit-Klassen. Insgesamt sind 32 verschiedene Klassen möglich. Der Name der Klasse ist nur für den Nutzer relevant und kann frei gewählt werden. Nach dem Namen folgt durch einen Doppelpunkt getrennt die Klassen-ID und anschließend, wieder durch einen Doppelpunkt getrennt, die Auflistung der Audit-Events. Events können mehreren Klassen zugeordnet werden.

Passt die Definition einer Klasse nicht auf eine Zeile muss dies durch eine Backslash gekennzeichnet werden. Die einzelnen Event müssen durch ein Komma getrennt werden.

```
# --------------------------------------------------------------------------
#
#     User Level Events and Event-Classes
#
#
#
# --------------------------------------------------------------------------

#     user events the nummer must be between 102 and 127, other events
#     will be ignored

AUE_LOGIN         : 100
AUE_LOGOUT        : 101
AUE_SU            : 102
AUE_AUDITON       : 103

#     Event Classes
#     Every unnamed event belongs to class AUC_NO

AUC_NO :  1 :
AUC_FR :  2 : AUE_OPEN_R, AUE_OPEN_RC, AUE_OPEN_RT, AUE_OPEN_RTC, \
              AUE_OPEN_RW, AUE_OPEN_RWC, AUE_OPEN_RWT, AUE_OPEN_RWTC, \
              AUE_USELIB
AUC_FW :  3 : AUE_OPEN_W, AUE_OPEN_WC, AUE_OPEN_WT, AUE_OPEN_WTC, \
              AUE_OPEN_RW, AUE_OPEN_RW, AUE_OPEN_RWT, AUE_OPEN_RWTC
AUC_FA :  4 : AUE_OPEN
AUC_FM :  5 : AUE_CHMOD, AUE_CHOWN, AUE_FCHOWN, AUE_LCHOWN, AUE_FCHMOD, \
              AUE_UTIMES, AUE_UTIME
AUC_FC :  6 : AUE_CREAT, AUE_LINK, AUE_MKNOD, AUE_SYMLINK, AUE_RENAME, \
              AUE_MKDIR, AUE_OPEN_RC, AUE_OPEN_RTC, AUE_OPEN_WC, \
              AUE_OPEN_WTC, AUE_OPEN_RWTC
AUC_FD :  7 : AUE_UNLINK, AUE_TRUNCATE, AUE_FTRUNCATE, AUE_RMDIR,
              AUE_OPEN_RT, AUE_OPEN_RTC, AUE_OPEN_WT, AUE_OPEN_WTC, \
              AUE_OPEN_RWT, AUE_OPEN_RWTC
AUC_CL :  8
AUC_PC :  9 : AUE_FORK, AUE_CHDIR, AUE_CHROOT, AUE_VFORK, AUE_SETGROUPS, \
              AUE_VTRACE, AUE_SETREUID, AUE_SETREGID, AUE_FCHDIR, \
              AUE_SETUID, AUE_SETGID, AUE_CLONE, AUE_SETFSUID, \
              AUE_SETFSGID AUE_SETRESUID, AUE_SETRESGID
AUC_NT : 10 : AUE_CONNECT, AUE_ACCEPT
AUC_IP : 11
AUC_NA : 12 : AUE_SYSTEMBOOT
AUC_AD : 13 : AUE_ACCT, AUE_SWAPON, AUE_SETHOSTNAME, AUE_SETTIMEOFDAY, \
              AUE_ADJTIME, AUE_SETDOMAINNAME, AUE_QUOTACTL, AUE_MOUNT, \
              AUE_STIME, AUE_SYSLOG, AUE_SWAPOFF, AUE_INIT_MODULE, \
```

```
               AUE_DELETE_MODULE, AUE_CAPSET
AUC_LO : 14 : AUE_LOGIN, AUE_LOGOUT
AUC_TF : 15
AUC_AP : 16 : AUE_SU
AUC_SL : 17
AUC_IF : 18
AUC_PR : 19
AUC_XA : 20
AUC_XX : 21
AUC_XC : 22
AUC_XD : 23
AUC_XF : 24
AUC_XM : 25
AUC_LP : 26
AUC_IO : 27 : AUE_IOPL, AUE_IOPERM
AUC_EX : 28 : AUE_EXECVE
AUC_AU : 29 : AUE_SETAUID, AUE_SETAUDIT, AUE_AUDIT, AUE_AUDITON
AUC_IL : 30
AUC_NF : 31 : AUE_NFS_SVC, AUE_NFS_GETFH, AUE_EXPORTFS
AUC_OT : 32
```

Wie im Beispiel ersichtlich, müssen alle Klassen aufgeschrieben werden, dabei ist es allerdings möglich, dass einzelne Klassen leer bleiben.

D.5 Konfiguration der Audit-ACL: `audit_acl.conf`

Die Datei `audit_acl.conf` enthält die Konfiguration der Audit-ACL. Alle in den Kernel eingefügten ACEs sind nur für die Laufzeit des Systems gültig. Die Konfigurationsdatei enthält alle Regeln, die bei jedem Neustart eingefügt werden sollen.

```
# ------------------------------------------------------------------------
#
#                 audit acl configuration file
#
#    Read the audit_acl.conf(5) manual page in order to understand the
#    options listed here.
#
# ------------------------------------------------------------------------


# Event Section
# target : event, value

evt   : any       : all, 3

evt   : aid = 101 : id = 12, 0

# ACL Section
# target : event, value, ressource

acl   : any       : id = 18, 3,     /proc

acl   : aid = 101 : id = 18, 2,     /home/%u/i686, /home/%u/i686/mmx, \
                                    /home/%u/lib, /home/%u/mmx; \
                    class = 12, 3,  /usr
acl   : gid = 100 : all, 3,         /home/%u/.netscape

acl   : aid = 0   : id = 18, 3,     /dev
```

Die Datei unterteilt sich in zwei Abschnitte, die Konfiguration der Preselektionsmasken und in die Festlegung der ACEs. Eine Regel für eine Preselektionsmaske wird mit dem Schlüsselwort `evt` eingeleitet, eine ACE-Regel mit `acl`.

Die verwendete Grammatik und deren Schlüsselwörter wurden bereits im Abschnitt C.3 beschrieben. Der einzige Unterschied liegt bei den verwendeten Trennzeichen und bei der Wiederholbarkeit einzelner Teilregeln. So kann eine ACE-Regel eine Liste von Event- und Pfad-Beschreibungen enthalten, deren Element durch eine Semikolon getrennt werden. Ebenso können mehrere Pfade, durch ein Komma getrennt, für eine Regel definiert werden.

Das Metasymbol `%u` steht für einen Nutzernamen. Immer wenn eine Regel eingefügt wird, wird das Symbol `%u` durch den gerade aktuellen Nutzernamen ersetzt. Damit wird es bspw. mit nur einer Regel möglich, für jeden Nutzer einer Gruppe eine Datei oder ein Verzeichnis seines home-Verzeichnisse von der Protokollierung auszuschließen.

Anhang E

Literaturverzeichnis

[1] Bodo Bauer, Michael Burghart, S.u.S.E. Linux 5.0 Installation, Konfiguration und erste Schritte. S.u.S.E. GmbH, Fürth 1997

[2] Corel Corporation, Corel's Linux Strategy, White Paper, Oktober 1999

[3] Helmut Herold, Lex und yacc: Lexikalische Analyse, Addison-Wesley, München, Paris 1992

[4] Beck M., Böhme H., Dziadzka M., Kunitz U., Magnus R., Schröter C., Verworrner D., Linux Kernel-Programmierung, Addison-Wesley,

[5] Patrick R. Gallagher, Jr., A Guide to Understanding Audit in Trusted Systems, Fort George G. Meade 1987

[6] Thomas Dapper, Carsten Dietrich, Bert Klöppel, Windows NT 4.0 im professionellen Einsatz, Carl Hanser Verlag, München, Wien, 1996

[7] Solaris Sources, Sun Microsystems, Inc.

[8] H.E.R.T. Audit Dämon, http://plan9.hert.org/projects/linux/auditd/

[9] Linux Basic Security Module Project, http://linux.bsm.sourceforge.net