VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REACTIVE AUDIT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JURAJ HLÍSTA
AUTHOR

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REAKTIVNÍ AUDIT
REACTIVE AUDIT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JURAJ HLÍSTA
AUTHOR

VEDOUCÍ PRÁCE — Doc.Dr.Ing. PETR HANÁČEK
SUPERVISOR

BRNO 2010

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací rozšíření auditu v Linuxu, kterým je reaktivní audit. Tento mechanizmus přináší novou funkcionalitu ve formě možnosti spouštění reakcí na určité události generované auditem. Reaktivní audit je implementován jako plugin a jeho použití je volitelné. Tato práce se zabývá také pluginem, který ukládá některé události a na základě jejich analýzy poskytuje časově závislé statistiky pro první plugin. Výsledkem je, že mechanizmus reaktivního auditu dokáže odhalovat také anomálie na základě poskytnutých statistik. Odhalení anomálie může také vést k vykonání určité reakce. Mechanizmus reaktivního auditu je dostatečně obecný pro to, aby mohl být využíván v různých situacích.

Abstract

The thesis deals with the proposal and the implementation of an extension for the audit system in Linux – the reactive audit. It brings a new functionality to the auditing in form of triggering reactions to certain audit events. The reactive audit is implemented within an audit plugin and its use is optional. Additionally, there is another plugin which stores some audit events and provides time-related statistics for the first plugin. As the result, the mechanism of the reactive audit does not only react to some audit events, it is also able to reveal anomalies according to the statistical information and set off the appropriate reactions. It is a fairly general mechanism that can be useful in various situations.

Klíčová slova

audit, kernel, systémové volání, auditovací pravidlo, analýza, bezpečnost

Keywords

audit, kernel, system call, audit rule, reaction, analysis, security

Citace

Juraj Hlísta: Reactive audit, diplomová práce, Brno, FIT VUT v Brně, 2010

Reactive audit

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením doc. Petra Hanáčka a Mgr. Miloslava Trmača z firmy Red Hat. Uviedol som všetky literárne pramene a publikáce, z ktorých som čerpal.

....
Juraj Hlísta
May 25, 2010

Poděkování

Rád by som na tomto mieste poďakoval Mgr. Miloslavovi Trmačovi z firmy Red Hat za poskytnuté rady a konzultácie a za ochotu a čas, ktorý mi pri tvorbe práce venoval.

Contents

Chapter 1

Introduction

Software developers have always had a need to check what is going on inside code so that they take advantage of debuggers to find bugs in their programs. This is just a low-level look at a running program. On the other side, system administrators have different observation needs that vary from those of developers – they require a high-level view of the system as a whole, therefore, the operating system auditing was developed.

Audit is useful for a number of reasons, it adds an ability of inspecting what actions have been performed by a particular user or a process, a system administrator can watch what decisions the system has taken. Furthermore, in case something went wrong, it can be investigated what actually happened or reveal some system misconfiguration from audit logs. There is an implementation of such audit in Linux operating system that is being developed mainly by Red Hat nowadays [19].

The primary task of the audit in Linux is to collect information about events occurring in the system. The audit is supported in the kernel [16] where it checks all the events and sends information about them to a user space audit daemon. Its function is to store the information to a log file and possibly pass the information to other user space tools for analysis. This is just a brief outline of how it works. The important thing to mention is that the kernel is supposed to check every single event and make a decision whether to generate an audit event, that is going to be sent to the audit daemon, or not. This is done by means of several filters that are a part of the audit subsystem in the kernel.

The filters determine if the audit events have to be generated according to audit rules. These rules are passed to the kernel from the user space [10] and can be changed (added, deleted) anytime by a system administrator, however, not automatically. It means, that the operating system is unable either to modify a set of rules or react to some events by executing commands on its own.

The main goal of this thesis was to develop an extension for the current audit system in Linux by implementing reactive audit and mechanism for creating statistics according to which reactions might be triggered. Chapter 2 discusses general audit requirements, the design of the audit in Linux, and auditing in other operating systems. The description of audit rules and audit events is given in the next chapter. The fourth chapter describes the mechanism of the reactive audit and its implementation is given in the fifth chapter. The following chapter, Chapter 6, discusses possible uses of the implemented mechanisms. Chapter 7 summarizes what was achieved and the last chapter concludes the thesis.

Chapter 2

Operating system auditing

This chapter discusses capabilities of audit mechanism as well as fundamental requirements that should be met in the audit system. It also aims at the design of the audit in Linux.

2.1 Audit capabilities

The audit in general was developed to increase the level of system security. It is achieved by reliable and configurable logging of a variety of security-relevant events, including configuration changes, login attempts, file access, etc. The audit does not log only actions, that were performed by a process or a user, but it keeps a track of unsuccessful attempts to perform an action as well. For instance, a user tried to open a file without a permission for reading. Despite the file was not opened, the audit recorded this event. Audit records can be used for system monitoring or analyzing of a number of different things, such as intrusion detection.

As regards Linux, it has got its own implementation of the audit mechanism, which is able to log miscellaneous types of events ranging from file operations to network related actions. A system administrator can choose what events are of interest and should be logged. Moreover, the events can be analyzed either in real-time or later – after they are stored in a log file. There is a number of tools, that are part of the audit mechanism. For example, it is possible to search certain events within an audit log file, make summary reports or analyze the audit events in real-time. Linux offers a similar mechanism for gathering information from the whole system which is called *Syslog* [20]. Although, the Syslog is mainly intended to record states of a system, such as hardware alerts. On the other side, applications can send logging messages to the Syslog but it is up to the applications what kind of data they send. They might not record everything. On the top of that, these messages are not delivered reliably, they might be dropped. The audit in Linux is a fairly general mechanism, which provides reliable logging of a large amount of various events. It can be configured and used in many different ways.

2.2 Audit requirements

To begin with, there are plenty of standards and norms in the field of information technology and various kinds of systems are designed to comply with them. Likewise, the audit system as a security component of Linux operating system was implemented to comply with a couple of security standards. Every standard describes requirements and an operating system can be certified only if its components (including the audit system) meet these requirements.

The audit was created primarily in response to *Common Criteria* (CC) [21]. This document specifies functional and assurance requirements related to security. Moreover, a process of evaluation is a part of CC, too. In other words, CC provides assurance, that the process of specification, implementation and evaluation of a computer security product, has been conducted in a rigorous and standard manner. *Evaluation Assurance Level* (EAL) is a numerical rating describing the depth and rigor of an evaluation. CC lists seven levels, with EAL1 being the most basic and EAL7 being the most stringent – *Red Hat Enterprise Linux 5* (RHEL5) gained EAL4 [23] (in some particular configuration and with some particular components). The audit system was one on the components that took part in the process of certification. Some of the requirements, that the audit satisfies according to CC, are:

- the audit is able to generate audit records of the following auditable events:
    - start up and shutdown of the audit system
    - attempts to import or export information
- each record is supposed to contain at least:
    - date and time of the event, type of the event, subject identity, outcome (success or failure of an event)
    - other relevant information for each audit event type
- it is possible to associate every auditable event with the identity of a user that caused the event
- it is possible to select a set of events to be audited from a set of all auditable events using audit rules, based on the following attributes:
    - object and subject identity
    - user and host identity
    - type of the event
- audit records can be read and interpreted in a manner suitable for a user
- ordinary users do not have access to audit records
- audit records are protected from unauthorized deletion or modification
- the audit has to ensure that audit records will be maintained when the following conditions occur:
    - audit storage exhaustion
    - failure
    - attack

One part of CC is *Controlled Access Protection Profile* (CAPP), which specifies a set of functional as well as assurance security requirements one of which is the audit capability. As for CAPP certified systems, they are assumed to be able to record security-relevant events, which affect safety and integrity of system's processes and data. In order to get a certification, the audit has to record information such as date and time of an event, type of an event, an identity of a user that caused the event, etc. The information that should be included in audit records is similar to the requirements of CC. Although, CAPP specifies other requirements that are not included in CC. They are referred to additional types of records the audit must support:

- all modifications to system configuration
- all use of authentication mechanism
- changes to any trusted database (`/etc/shadow`)

2.3 Design of audit in Linux

There are two main parts, which the audit system can be split into, when talking about the overall design. The main audit components are depicted in the picture 2.1. One part is situated in the kernel, whereas the second is implemented in the user space [16]. There might arise a question – why is the audit system divided between the user space part and the kernel part?

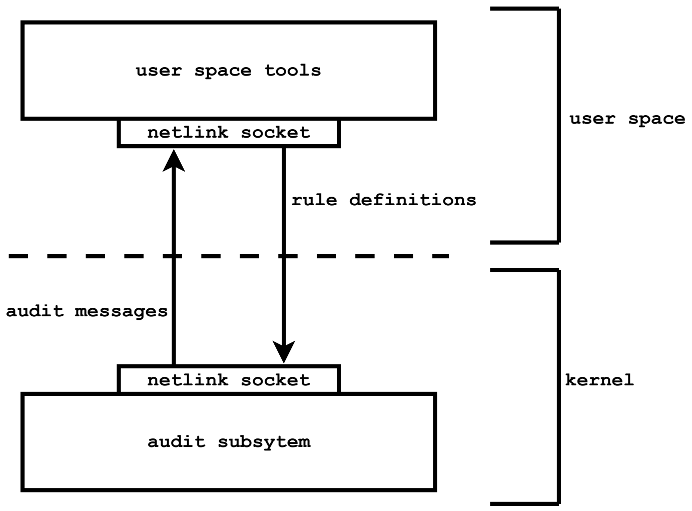


Figure 2.1: Overall design of the audit system.

The answer for the question would be, that the audit has to be a part of the kernel, because it is possible to observe and check all events in it. On the other side, the events can be monitored from the user space, too. However, this approach would have to deal with more difficulties, for instance, *time-of-check-to-time-of-use race conditions* [1] could arise. They occur when a resource is checked for a particular value, that value is changed, then the resource is used, based on the assumption, that the value is still the same as it was at the check time. Provided that, a user space application is being monitored from the user space, the application calls a function `open(buf,...)` and a value `buf=/dev/null` is audited. After that, the value may be rewritten to something else (such as "`/etc/shadow`"). It happens before a *system call* (see subsection 2.3.1) is entered. It results in a problem, because a different value is audited than the value which is used within the system call. It could be solved in the user space, although, overhead cost would be high. Checking the events in the kernel eliminates this kind of problems. It is more efficient and not so complicated.

The kernel does not contain the whole implementation of the audit mechanism, even if it is theoretically possible. The kernel is supposed to provide mechanism, not policy. It means, that it monitors what is going on in the system and sends audit events to the user space. It does not call for what way the audit events should be processed or where configuration files should be located. Besides, the user space is much more friendly programming environment with lots of libraries, an opportunity to choose a programming language, "unlimited" virtual memory, etc.

Taking the above facts into the consideration, the audit system in Linux had to be divided into the kernel and the user space. It checks what is going on and generates audit events in the kernel, whereas user space tools are responsible for logging and analyzing the audit events and configuring the audit. These two parts are mutually dependent.

2.3.1 Types of events

The audit mechanism distinguishes between two the types of audit events. Before going ahead talking about the events, it is important to understand what the *system call* [17] is. It is a mechanism in Linux used by an application program to request a service from the operating system. The services, that the operating system provides, are represented by functions which are available at the user space level. There is a number of system calls, for instance, in order to create a new file, function `creat()` is called. System calls provide an interface between a process and the operating system.

Having realized what the system calls are, the audit events can be divided into two categories [16]:

- **system call events** – allows recording whenever the kernel leaves a system call. Basically, there are two types of filters (shown in the picture 3.1) in the kernel, that control system call events.
- **other types of events** – this category consists of events, that are handled separately. They are triggered by system calls, but they are not considered to be system calls. For example, a user is trying to log in which can cause that a system call will be triggered. After that, however, an audit event carrying an information whether the login was successful or not is generated.

In spite of the fact the system call auditing is a significant part of the whole audit, it can be turned off, which increases system performance. It is because the system calls are quite frequent and collecting relevant information for every system call has some impact on the system performance.

2.3.2 Audit subsystem

The audit subsystem is a part of the audit that is situated in the kernel. Since it is placed in the kernel, its function is supposed to be essential – it checks all events originating in an operating system and generates audit events that are sent to a user space daemon. In case the user space daemon is not in use, the audit subsystem must work anyway (the audit events are sent to the Syslog (see section 2.1)) or the operating system might be even stopped – it depends on a configuration. The audit events are forwarded to the user space where they are stored and analyzed. No analysis is done inside the kernel at all.

As for the implementation of the audit subsystem, there are several components. First of all, the subsystem involves a rule database where audit rules are stored. They are passed to the kernel from the user space. The rule database consists of six rule lists – each list keeps a different type of the audit rules [16]:

- **entry** – apply a rule at syscall entry.
- **exit** – apply a rule at syscall exit.
- **user** – apply a rule to events originating in the user space – generated, formatted and sent from the user space.
- **task** – apply a rule at the time of a task creation – when `fork()` or `clone()` is called by a parent task.
- **watch** – apply a rule to file system watches
- **type** – apply a rule when an audit event is created – it decides whether the audit event will be sent to the user space

The type of the rule denotes when it is supposed to be activated. Furthermore, another component of the audit subsystem is a set of filters. Their task is to control all the events and match them against the audit rules stored in the lists. Each filter is tied together with its corresponding list, hence, there are six filters as well. These six lists and filters can be considered as the same component, for the purpose of clearing it up, they were studied as two different things here. When a match with some rule is found and the rule does not prohibit auditing, an audit event is created and sent to a user space audit daemon.

The kernel communicates with the user space and vice versa by means of *netlink socket* [4]. It is a method of communication of processes and the kernel with each other. The main advantage of the netlink socket is a standard socket interface in the user space. The netlink socket is datagram oriented, it contains several protocols and others can be added as well. This kind of communication between the kernel and the user space is used because it provides an ability to send datagrams to and from the kernel – it is principally important for auditing. There is also the *ioctl* [17] which is used for passing messages to the kernel. Although, there is no convenient way of sending messages from the kernel to the user space.

When it comes to the system call auditing, processes are monitored. Every process in the kernel is represented by an instance of a structure having certain information about the process. For the purpose of auditing, the structure was extended with the *audit context* [16] to keep auxiliary data relevant for a system call. These data are collected from different parts of the system during execution of the system call. Each auxiliary information causes a generation of a new audit record, so that one audit event can be composed of more audit records. They are joined together with the same time stamps and serial numbers. Format of the audit events is described in the section 3.3. The system call auditing utilizes entry and exit filters that work differently. When the kernel leaves a system call, it matches collected (auxiliary) information with the rules stored in the rule database (the exit list). If a match is found, it generates an audit event (one or more audit records according to the number of auxiliary data). This is how the exit system call auditing works. The entry system call auditing is able to make a decision if the auxiliary data should be collected at the system call entry (the entry filter is used). Therefore, it is faster when the auxiliary data are not collected and then checked.

2.3.3 Auditd

It is the user space daemon able to communicate with the kernel via the netlink socket. Its task is simple – just to read audit events that have been produced by the kernel as fast as possible and store them in an audit log file. It does not do any translation or changes to the audit events [11]. Additionally, the daemon distributes the audit events to the `audispd`. The picture 2.2 shows connections among the user space tools and the audit subsystem.

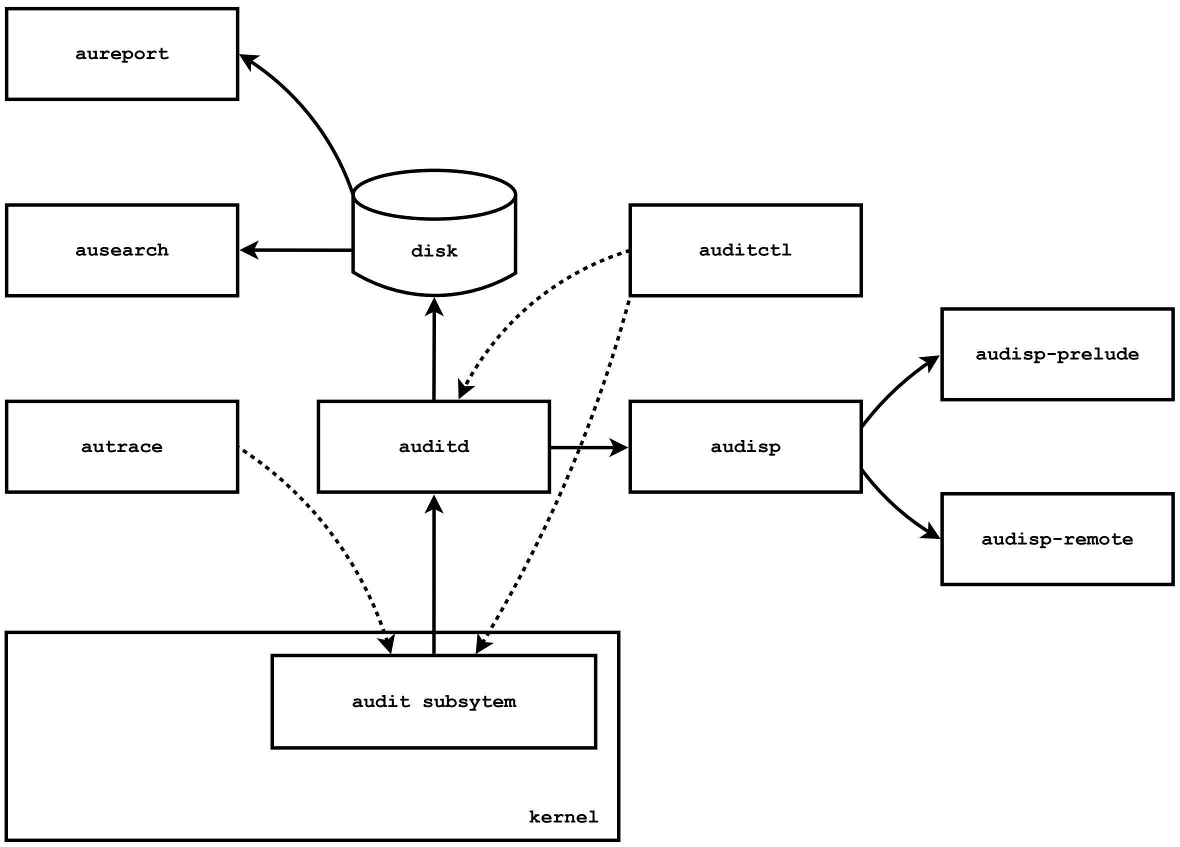


Figure 2.2: Connections among the user space tools and the audit subsystem.

2.3.4 Auditctl

This is an utility that controls the rule database and sets a couple of parameters of the audit subsystem from the user space. It is possible to add or delete the audit rules, temporarily disable or re-enable auditing, set what action the kernel should take in case of a failure, etc. Furthermore, the `auditctl` is used to check the content of the rule database or the status of the audit subsystem [10].

2.3.5 Audispd

The `audispd` is an audit event multiplexor or it is called the event dispatcher as well. It has to be started by the `auditd` in order to get audit events, that are distributed to child programs (plugins). The audit events may be analyzed within the plugins in real-time [8].

2.3.6 Ausearch

This user space tool is a preferred way of how to look at audit logs. It is able to interpret audit messages saved in a log file to human readable form. In addition, it is possible to use simple queries on the audit logs and extract only audit events, that are of interest [13].

2.3.7 Aureport

It is a tool which produces summary reports from the audit logs. The reports produced by the `aureport` may be used as building blocks for more complicated analysis. Moreover, this tool can be joined together with the `ausearch` and create reports based on some extracted data [12].

2.3.8 Autrace

It is a program similar to the *strace*. Because the audit is able to monitor system calls, the `autrace` takes advantage of it and adds audit rules that will record system calls of some process. This command deletes all audit rules prior to executing the target program and after executing it [14].

2.3.9 Plugins

Plugins are part of the audit system, too. However, their use is optional. They extend basic functionality of the audit system. There is a couple of different plugins such as `audisp-prelude`, `audisp-remote` and `audisp-zos-remote`. The first of them analyzes audit events in real-time and sends detected events to the `prelude-manager` for correlation, recording and to be displayed [6]. The latter ones, the `audisp-remote` and the `audispd-zos-remote`, are alike. They perform remote logging to an aggregate logging server [7, 9].

In order to parse audit events, the plugins can take the advantage of the `auparse` library [3]. The plugins, that do not analyze audit events, such as `audispd-remote` and `audispd-zos-remote`, only forward what they receive without any parsing. The library provides an interface with a number of functions and internal data structures that simplifies its use in programming languages.

This work describes the proposal and the implementation of new plugins (`audispd-reactive` and `audisp-stats`) that are able to analyze the audit events in real-time, make statistics and react to some of the events by changing the rule database in the kernel or executing commands. The plugins are described in more detail in the following chapters.

2.4 Audit in other operating systems

The Linux implementation of the audit mechanism is not the only one. There are also other operating systems with their own implementations, such as FreeBSD and Windows NT.

2.4.1 FreeBSD

FreeBSD's audit is a part of the TrustedBSD project that is developing advanced security features for the FreeBSD operating system, including file system extended attributes and UFS2, access control lists, OpenPAM, security event auditing with OpenBSM, mandatory access control, etc. Many technologies from TrustedBSD may also be found in operating systems beyond FreeBSD, including Mac OS X, NetBSD, OpenBSD, and Linux. This project was established due to requirements of the Common Criteria [22].

As for the implementation of the FreeBSD's audit, it consists principially of the following components:

- **sys/security/audit** – reliable kernel audit record queue, system call auditing.
- **contrib/openbsm** – BSM API library, documentation and audit-related utilities.
- **etc/security** – configuration files.
- **usr.sbin/auditd** – audit management daemon.

Audit record queueing and audit event capture for the majority of auditable events occurs in the kernel. In addition to audit records generated in the kernel, trusted user applications may also submit to the kernel using the `audit()` system call. Review and management of the audit records (referred to as audit trail) is performed using user space applications [22].

The format chosen for the audit trail is defined by a standard as part of the Basic Security Module (BSM). The BSM record format is designed to be independent of processor architecture, and may be easily extended to support features present in FreeBSD. The BSM log consists of one or more audit records. Each record is composed of a series of tokens representing data elements, with every record containing header, subject, return, and trailer as is shown in the picture 2.3. The header token contains general information about the

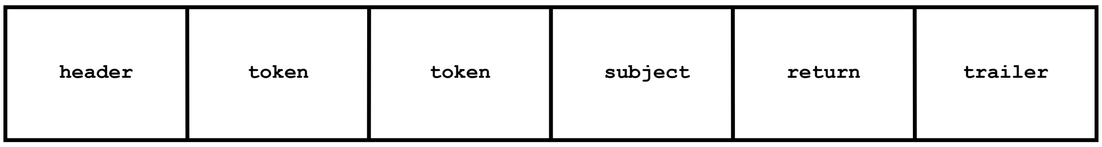


Figure 2.3: Audit record format in FreeBSD.

audit event (record length, event type, timestamp), the subject token contains user id, pid, etc. The return token contains the system call return value as well as the success or failure indication. The trailer token closes out the audit record and stores the entire length [22].

Audit events are associated with system calls. There are *event classes* such as *process* or *network* that describe various categories of similar audit events. The kernel maintains an internal table that maps events to the classes and this table can be changed from the user space. Communication from the user space to the kernel is done via system calls, while messages from the kernel are sent to the user space via the `/dev/audit` special file [22].

In order to fully support the audit requirements, several user space programs are part of the audit system. The core application is the audit daemon (`auditd`) responsible for audit configuration management audit trail files. The audit trails are stored in BSM binary format, so that user space tools must be used to read the trails. The `praudit` command converts trail files to a simple text format and `auditreduce` may be used to reduce the audit trail for analysis, archiving, or printing purposes [22].

2.4.2 Windows NT

The *Event log* is the Windows monitoring and reporting mechanism that collects *event messages*. Each message includes information such as a timestamp, a message type, an event ID, the source of the event, and other message-specific data. Each event log message has some static data common to every message with the same event ID, and a set of data that changes from one message instance to another [2].

There are three event logs by default:

- **application**
- **system**
- **security**

The Application log is used by applications to log the events specific to them. Each application logs events under a different source name, that could be used for filtering to see only messages logged by a particular application. The system log is used by system services such as DHCP. The security log is where audit messages, such as logon success, logon failure, and object access, are stored. The application and system logs contain three different types of messages:

- **error**
- **warning**
- **information**

The security log, however, has very different types of messages. It contains only success or failure indication [2].

The application event log has weak default protections. The system log has stronger controls over who can write to the log, but it does not place many restrictions on who can read the log. Any user can read both application and system log and can write to the application log, therefore, the application log is not trustworthy. The Security log is unlike the other logs in two aspects. Firstly, in the default configuration it is protected by an access control list (ACL) and privilege checks, which limits the set of users who can read its contents to administrators and holders of the security privilege. Secondly, only one entity is allowed to write to the security log – the Local security authority (LSA). This design ensures that the security log contains information only from trusted sources [2].

Chapter 3

Audit rules

The overview of audit rules that the current implementation of the audit system includes as well as the description of audit events and a user space tool for adding and deleting audit rules is given in this chapter.

3.1 Placing constraints on events

There is a large number of events occurring in an operating system and it is not convenient to log them all for a number of reasons. Firstly, system performance would be decreased by a significant amount, the most of time an operating system would spend logging audit events. Secondly, a log file would require a lot of space to store such amount of information. Finally, the log file would contain a vast majority of the audit events which are not interesting from a point of view of a system administrator.

Because of that, there are audit rules which place constraints on the events, describe what events are of interest, ought to be logged and analyzed. Auditing requires some processor time to gather audit data from the system and check the events. That is why, a number of the audit rules being in use should not be high. Otherwise, it can have some negative impact on the system performance. Too many rules means lots of matching of each event against the audit rules.

3.2 Audit rules and auditctl

This section discusses the format of the audit rules in connection with the user space tool `auditctl`. Its primary purpose is to add and delete the audit rules and change the content of the rule database in the kernel. Each audit rule is composed of a number of parameters that vary according to what constraints are being placed on the events.

The very basic information is given by the *list/action* pair. The *list* denotes a type of a list that a rule will be appended to or removed from. It is important to notice, that internal representation of the rules in the kernel consists of six lists (see subsection 2.3.2), while the `auditctl` provides only five types of lists for the rules [10]. On the top of that, `entry` list is deprecated and not used anymore, the `exit` list is applied instead.

- `entry` – syscall entry (deprecated)

- `exit` – syscall exit
- `task` – task creation
- `user` – user-originated events
- `exclude` – events, that are not meant to be logged

The *action* can either be `never` or `always`. The latter choice enables the rule. The rules in the lists are checked one by one. If the rule's *action* is set to `never`, generation of audit events is suppressed in this list (the rules are not used when audit events are checked) – it applies for the rule with the `never` action and rules that are placed after this rule in the list.

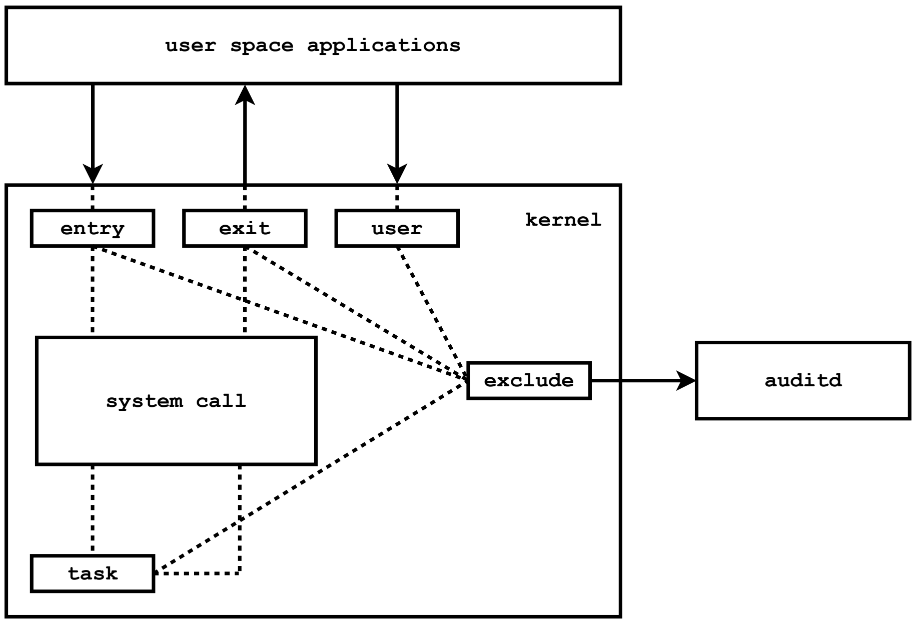


Figure 3.1: Kernel and various types of filters.

In spite of the fact, that a set of all events was divided into smaller groups, it is still not acceptable, because these groups are quite broad, each covering too many events. More constraints need to be put into the rules to select smaller subsets of the events. This is done by defining one or more *field/comparator/value* pairs. The *field* represents an attribute of an event which can be observed by the kernel. For example, it can be a process identifier, a device number, an argument to a system call, etc. The *comparator*(equals, more than, more or equal, etc.) and the *value* specify, which set of values the *field* is allowed to possess to trigger an audit event. As regards the system call auditing, it is also possible to add other restriction to specific system calls with the `-S` parameter followed by a name (or a number) of the system call, such as `open`, `creat`, etc. Due to the ability of adding more constraints like these to an audit rule, it makes the rule concrete enough to select the desired set of the events.

The general way of how to put new rules into the kernel or erase them is using the `auditctl` command followed by the following parameters:

```
-[a|d] filter,action [[-S syscall]*|[-S all]] [-F field[=|>=|>|<=|<|!=]value]*
```

There is also a possibility to make use of a different syntax when adding or deleting watches for file system objects:

```
-[w|W] path [-p [r|w|x|a]]
```

The parameter `-w` implies that a rule will be added. In order to remove the rule, the `-W` parameter is used. Another parameter, the `-p`, stands for permissions that should be filtered. The `auditctl` has other parameters, but it is not intended to discuss them all. There are manual pages [10] for more details.

3.3 Audit events

Audit events are generated by the audit subsystem. Their creation is triggered when an event that originated in the system was matched with an audit rule successfully. After the audit event is created, it is sent to the netlink socket and read by the `auditd` on the other side. The audit daemon does not make any significant changes to the audit events, it just stores them to a log file and forwards them to the `audispd`.

For instance, putting the following audit rule into the rule database:

```
  auditctl -a exit,always -S open -F path=/tmp/file
```

means that an audit event will be generated, if there is an attempt to open the file `/tmp/file`. Thus, the command `cat /tmp/file` causes that the following audit event is generated:

```
type=SYSCALL msg=audit(1261834905.528:4): arch=c000003e syscall=2
success=yes exit=3 a0=7fff330fc9a7 a1=0 a2=2 a3=0 items=1 ppid=1169
pid=1223 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=pts1 ses=4294967295 comm="cat" exe="/bin/cat" key=(null)

type=CWD msg=audit(1261834905.528:4): cwd="/root"

type=PATH msg=audit(1261834905.528:4): item=0 name="/tmp/file" inode=27872
dev=03:01 mode=0100644 ouid=0 ogid=0 rdev=00:00
```

This audit event is composed of three audit records. The first record carries the information about the system call and the rest of the records includes auxiliary information relevant for the system call. As regards the format, every audit event has its type, a time stamp and a serial number. All other information is specific to the type of the audit event. Every audit record can be divided into audit fields. The audit field has its name on the left and value on the right side. The audit records are joined together based according the same time stamp and the serial number forming the audit event. In this example, there is one audit event composed of three audit records.

Chapter 4

Reactive audit

This chapter deals with a new extension of the audit mechanism – the reactive audit. The explanation why the reactive audit is beneficial and the description of how it is supposed to work is given here. In addition, the format of reaction definitions is discussed in the last part of this chapter.

4.1 Benefits of reactive audit

In the first place, it is necessary to explain how the reactive audit is supposed to work so that its benefits can be given. As the name implies, it can react to certain events or states of a system which are retrieved from audit events. The reactive audit can be implemented as the audit plugin – it receives the same audit events as the `auditd`. The plugin must be able to recognize when a reaction should be triggered based on information from audit events. For this purpose, a language that describes conditions when a reaction is activated, must be created. The reaction results in adding or deleting audit rules or executing commands.

The reactive audit may be used to reduce the number of audit rules stored in the rule database. It could have some positive impact on the system performance due to less rules would be matched with the events originating in a system. For instance, there are many users in a system and a system administrator uses different sets of rules for each of them to monitor their activities. Without the reactive audit, the rule database would have to contain all the audit rules specific for each user. If the reactive audit was in use, it would reduce the number of the audit rules kept in the rule database significantly. When a certain user logs in, the reaction that adds a set of rules specific for the user is performed. Analogically, when the user logs out, the set of rules will be removed. When it comes to file system auditing, the reactive audit might watch for an audit event indicating that a USB stick is attached to a computer. As the reaction, new rules which monitor what is sent to or from the USB stick may be added. In general, using the reactive audit, it is possible to reflect the current state of a system by keeping only necessary set of rules within the kernel.

Another example describes a situation, when warnings are sent to a system administrator by the mechanism of the reactive audit. Supposing that, there is a SSH server running and users try to log in. The audit generates audit events about login attempts, that are analyzed. A reaction of sending an e-mail to a system administrator can be triggered, when a particular user attempts to log in unsuccessfully more times in a row. It can be done since the reaction definition can include an execution of a command.

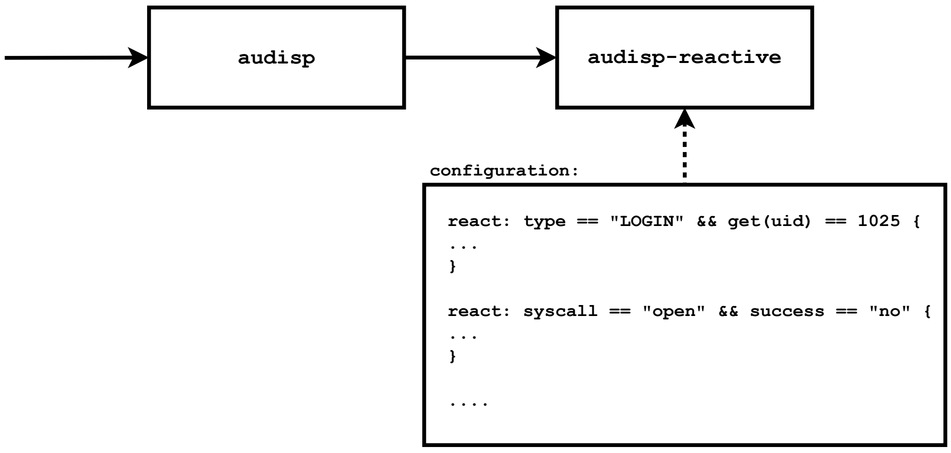
Furthermore, a system might protect itself taking advantage of the reactive audit. For example, an ordinary user tries to open one or more files that can be accessed only by a system administrator. In case the number of the attempts to open the important files reach some predefined number, the reaction of executing the appropriate command to log out or even ban the user from the system is set off. This kind of protection can terminate suspicious processes as well.

Considering the above examples, the mechanism of the reactive audit can be used in the following ways:

- reduction the rule set in the kernel, that reflects actual state of a system
- reaction to specific events by sending warnings to a system administrator
- protection of a system from suspicious activities

4.2 Audisp-reactive

There are two main ways of implementing the mechanism of the reactive audit. Both have their pros and cons. The first approach does not require any modifications to the existing code, it implements the whole mechanism in the plugin. A configuration file of the plugin defines conditions that should be met in order to trigger a reaction. The plugin receives audit events and analyses them all – it includes parsing every audit event to extract necessary information. Based on the configuration and the information gained from audit events the plugin makes a decision whether a reaction should be set off or not. The plugin utilizes `auparse` library for parsing the audit events. The configuration file defines the syntax (shown in the picture 4.1) that describes certain audit events which the plugin reacts to.



```
react: type == "LOGIN" && get(uid) == 1025 {
...
}

react: syscall == "open" && success == "no" {
...
}

....
```

Figure 4.1: Reactive audit implemented only within the plugin.

Another approach depends on some support for *reactive rules* right in the kernel. The reactive rule extends the audit rule with a new field (`react`) that is similar to the `key` field. It carries an identifier of a reaction that is defined in the plugin's configuration file. This information is passed to the kernel. When a match with this rule is found, the kernel generates a new type of an audit record. The audit record is composed only of reaction identifiers that the plugin watches for and set off reactions accordingly. The plugin does not perform any analysis, it parser only the new type of the audit record (shown in the picture 4.2). This approach is faster than the first one, but has many drawbacks. First of all, it needs the kernel as well as some user space tools to be modified, especially `auditctl` to support the reactive rules. Furthermore, the reaction identifiers are the only way of how to recognize when a reaction should be triggered.

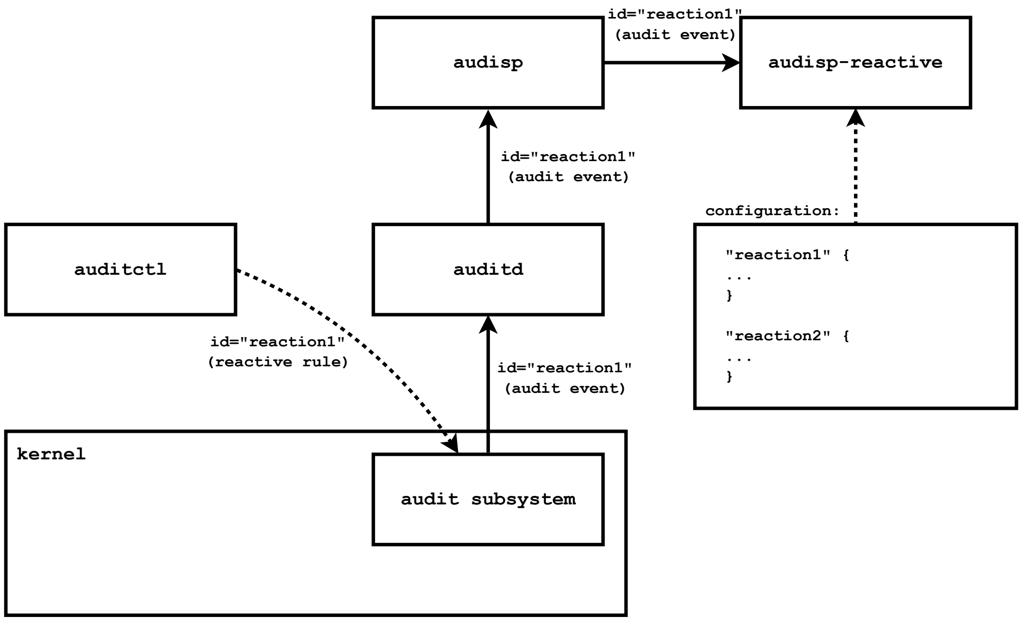


Figure 4.2: Reactive rules and their support in the kernel.

The second approach brings a lot of changes to the existing code. Even if the plugin would be able to react faster without parsing every audit event, it is not flexible enough. It is because the reactions only describe certain audit rules, that have been found a match with using the reaction identifiers. In spite of the fact, that the first approach means parsing every audit event, it is more configurable, flexible and it does not require any modification to the existing code to support the reactive audit. Since the first approach comes up with more advantages that prevail the dissadvantages, it was chosen to be implemented.

4.3 Defining reactions

The configuration file of the `audispd-reactive` contains definitions of reactions to different audit events. The file is read when the audit starts. Even if the plugin is in use, the file can be changed and a new configuration is read without stopping the plugin, let alone the `auditd`. The file can be divided into two main parts. The first one involves definitions of variables and constants that are used in the second part, where definitions of reactions are placed.

A simple programming language is used within the configuration file. It is a weakly typed language – it can be defined as a mixture of a scripting language (such as Python) and the

C programming language. Despite the fact that there are programming languages such as Perl, Lua, Tcl and Python which could be used in the configuration file, it is a better solution to develop a new language for a number of reasons. The existing languages are too complex, they offer plenty of programming structures and their syntax can be complicated. Although, the plugin's configuration is meant to be simple and specialized, it does not require such complicated language at all.

This is an informal description of the language, especially its syntax. The configuration of the plugin consists of several basic elements:

- constants and variables
- assignment and basic arithmetic operators
- conditions (conditional and logical operators)
- commands for adding and deleting audit rules
- command for executing tasks
- functions for retrieving information from audit events

There are also comments, each comment starts with "#" character and ends with the end of the line. No loops are put into the language, there is no use complicating it. Moreover, a mistake could occur within a loop, making it endless, which would cause a failure of the plugin.

Definitions of global variables and constants are placed outside of the reaction definitions. The global variables and constants can only represent integer values (positive and negative) and strings. The syntax, that is used to define a string or an integer variable (constant) is the same for both types:

```
var identifier = value
var identifier = "value"
```

The same syntax applies to the constant with the difference that there is the keyword `const` instead of the `var`.

String values are enclosed in double quotation marks. As for integer values and integer identifiers in connection with different operations, it is allowed to perfrom basic arithmetic operations with them (addition, subtraction, multiplication, division and modulo):

`[identifier|value] [[+|-|*|/|%] [identifier|value]]`$^{+}$

String values and string identifiers are different in this way, they can only be concatenated, although, it is important, that concatenation applies even among strings and integers. There must be at least one string value or string identifier within the expression:

`[identifier|value|"value"] [+ [identifier|value|"value"]]`$^{+}$

Assignment operator works for both integers and strings in the same way:

```
identifier = [identifier|value|"value"]
```

There are also conditional operators ("equals", "less than", "less or equal", etc.) – all of them can be used with integer values and integer identifiers:

```
[identifier|value] [==|>|>=|<|<=|!=] [identifier|value]
```

However, only "equals" and "not equal" operators apply for string values and string identifiers:

```
[identifier|"value"] [==|!=] [identifier|"value"]
```

In addition, there are logical operators (and, or, not) used within a condition expression and apply for integer values and integer identifiers only:

```
[identifier|value] [&&|||] [identifier|value]
![identifier|value]
```

The very basic structure of the language is the conditional statement. It has the following syntax:

```
if (condition) {...}
[else if (condition) {...}]*
[else {...}]
```

Conditional statements use curly brackets ("`{`" and "`}`") to form blocks. However, if there is just one expression (command) inside the block, the curly brackets are not necessary (just like in the C programming language).

The reaction definition starts with the keyword "`react:`". It is split into a condition and an action part. The condition is composed of comparison and logical operators as well as global identifiers, string literals and integers. It forms a conditional expression. The action part is put inside a block that includes what is actually done as a reaction to a certain audit event. It can contain a definition of a local variable or constant, a conditional statement, an assignment and logical expression and commands for adding (deleting) rules and executing shell commands:

```
react: condition {...}
```

The command for adding an audit rule has the same syntax as the `auditctl` command. The only difference is that the keyword `add` stands for the `-a` parameter (which is analogous to the `del` and the `-d`):

```
add "filter,action [[-S syscall]*|[-S all]] [-F field[=|>=|>|<=|<|!=]value]*"
del "filter,action [[-S syscall]*|[-S all]] [-F field[=|>=|>|<=|<|!=]value]*"
```

Because the `auditctl` supports different syntax for adding a deleting file system watches, there are other two commands defined by the keywords `addw` and `delw` (the "`w`" stands for "watch"). The former one replaces the `-w` parameter and the latter one the `-W` parameter of the `auditctl` command:

```
addw path [-p [r|w|x|a]]
delw path [-p [r|w|x|a]]
```

For the purpose of executing different tasks, there is the `exec` command:

```
exec "[task|path] [parameters]"
```

Besides these commands, the language also provides the `get()` fucntion. It is a fundamental element of the language as the function extracts information from the current audit event being processed. It requires one parameter – field name, such as “`uid`”, “`gid`”, “`key`”, etc. It returns the value of the specified field. The function can be used within the condition as well as the action part of the reaction definition. This makes the language flexible enough, because a reaction can be set off according to what values are returned from the current audit event. Moreover, the action part of a reaction might take advantage of the `get()` function and form a command using string concatenation. For example, it is possible to add a new rule as follows:

```
add "exit,always -S open -F uid=" + get(uid) + " -F success!=0"
```

There is another function – the `getq()` that returns a field value that is wrapped into quotes. This function is meant to be used in connection with the `exec` command to reduce security risk of executing dangerous code.

To sum it up, the programming language used in connection with the configuration file might seem to be too simple to be worthy. Despite it provides only three commands and functions for extracting field values, the `exec` is able to execute any command in a system. The plugin can be used in different ways, that makes this mechanism usable in many scenarios. The section 4.1 describes some examples in which the reactive audit takes part.

4.4 Audit statistics

Not only the audit log file stores audit events, but also the SQL database can be used for this purpose. It does not have to include a complete information about each audit event, only important fields are sufficient (“`uid`”, “`gid`”, “`pid`”, etc.). Likewise, not all the audit events are of interest, therefore, only those the plugin stores that have the `key` field set to “`stats`”. The audit plugin called `audispd-stats` receives the audit events and puts them into the database. Moreover, it provides an interface to access the database for other programs or plugins, especially `audispd-reactive` (shown in a picture 4.3). Since the reactive plugin has an access to the database, it can trigger a reaction based on some statistic information. The database includes data about users or processes and actions that have been performed for the last day, week, etc. In other words, it can be determined whether a current action deviates from the actions included in the database. For example, a certain user logs in once or twice a day which is recorded in the database. In case the user logs in more than four or five times during a day, this event will be marked as an *anomaly* and the appropriate reaction will be set off.

4.4.1 Audisp-stats

The plugin utilizes SQL database where field values and their time stamps are stored. They are not put into a regular file, because it is not possible to search within the file using complicated queries and make statistics. In addition, it is much faster to search data in a database than in an ordinary file. A configuration file specifies what fields of the audit events should be stored, therefore, the audit events must be parsed to select only the described fields using the `auparse` library.

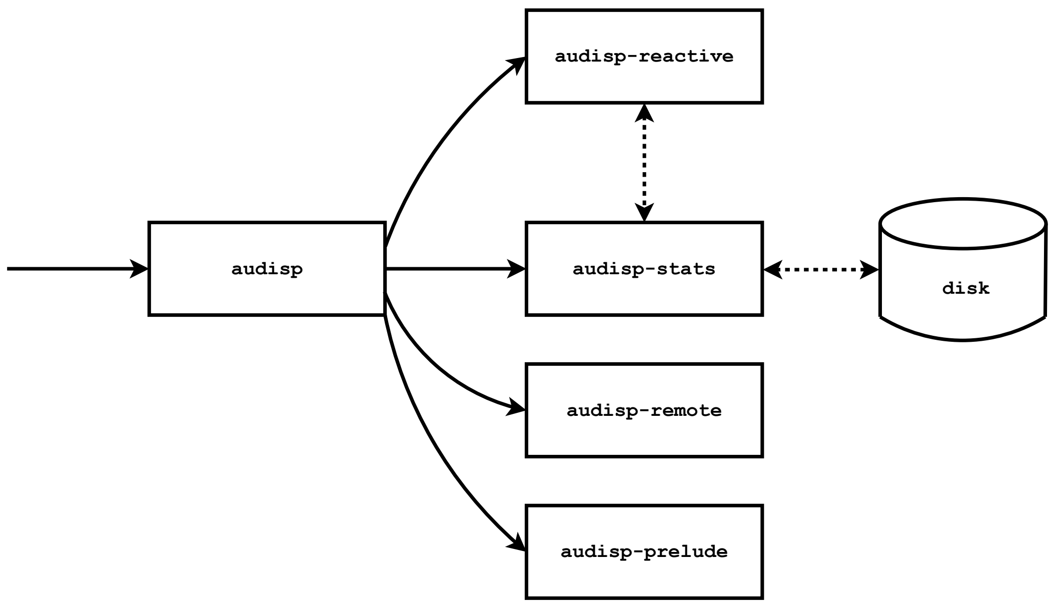


Figure 4.3: Interaction of the reactive plugin and the plugin for storing audit events.

The interface that this plugin provides for accessing the database is simple, it is one function with three parameters. The first one places conditions on the fields stored in the database using comparison and logical operators. The second parameter is a starting point in the past and the third is the end point. The two parameters represent a time interval. When the function is called, it always returns the number which denotes how many times the specified event by the first parameter appears within the time interval specified by the second and the third parameter. The following example shows how a statistical information may be retrieved from the `audisp-reactive`:

```
stats("uid=1000 AND type='LOGIN' AND success='no'", 1 week, now);
```

The `stats()` function returns the number of unsuccessful login attempts of a user with the `uid` 1000 for the last week.

4.4.2 Anomalies

Keeping the track of the audit events in the database allows the reactive audit to detect anomalies. Because the audit has the information about actions that occurred in the past, it can “expect” actions that will happen in the near future. If a particular action is originating in the system more or less frequently than in the past, it is considered as the anomaly. The anomalies might be threats for a system. For example, if the number of unsuccessful logins is higher than it used to be, it is the anomaly that might indicate that someone unprivileged is trying to get to the system. The reactive audit is supposed to send a warning to a system administrator or trigger other appropriate reaction.

Definitions of reactions can take advantage of the interface provided by the `audispd-stats` plugin. It is possible to detect anomalies inside the reactin definitions and react to them as soon as they are detected. If the last example is extended with another calling of the same function with the same first parameter but different time interval, there will be two numbers referring to how many times the same event occurred within different time periods:

```
x = stats("uid=1000 AND type='LOGIN' AND success='no'", 2 week, 1 week);
y = stats("uid=1000 AND type='LOGIN' AND success='no'", 1 week, now);

# detection of the anomaly
if (y > x + 50) {
  ...
}
```

If the two numbers deviates in some way defined in the configuration file of the reactive plugin, it is considered as the anomaly.

Chapter 5

Implementation

The detailed look at how the `audisp-reactive` and the `audisp-stats` are implemented is given in this chapter. Both plugins are written in the C programming language.

5.1 Audisp-reactive

This is the main plugin that implements the mechanism of the reactive audit. It can perform analysis and set off reactions either with the support of the `audisp-stats` or independently.

5.1.1 Configuration and abstract syntax tree

Since the `audisp-reactive`'s configuration file comes up with the new programming language, it can be read and parsed using *Flex* and *Bison* which are tools for building programs that handle structured input. Flex is a lexical scanner, that reads the configuration file and sends lexical tokens to the Bison parser. The parser contains a set of rules that are used to turn a sequence of tokens into a *parse tree*. The Bison rules are basically *Backus-Naur Form* (BNF), with the punctuation simplified a little to make them easier to type. BNF is a standard form of how to write down a *context-free grammar* (CFG). There are various subclasses of CFG grammars, although, the Bison parser is optimized for *lookahead left to right parsing* (LALR). It is a type of bottom-up parser, which attempts to build the parse tree from the bottom of the tree upward toward the starting symbol. The definition of the grammar that is used in the configuration file is left recursive. It is because the parser works more efficiently with this grammar in comparison with the right recursive grammar [5].

The reactive plugin requires the data structure that can be used to represent reaction definitions in the configuration file. The appropriate candidate is the *abstract syntax tree* (AST) [5]. The AST is basically a simplified parse tree – it does not contain every detail that appears in the real syntax. For example, no parentheses are included in the AST. The AST can be evaluated easily, and reset to the original state. After the tree is reset, it can be evaluated again with the same result. It is analogous to the reactions because they are likely to be set off more times. As the Bison parser builds the parse tree, it constructs the AST as well. The two trees are created simultaneously. The building blocks of the AST are AST nodes.

Every AST node is of the type `struct ast` which has the following definition:

```
struct ast
{
   int line;                 /* line number in config file */
   int processed;            /* states if the node was evaluated */
   enum node_type type;      /* node type */
   struct ast_value value;   /* node value */
   struct ast *x;            /* pointer to left-side node */
   union {
      struct {
         struct ast *y;      /* pointer to right-side node */
      } node;
      struct {
         struct symbol *sym; /* symbol reference */
      } ref;
      struct {
         char *f;
         int q;              /* quotes */
      } field;
      struct {
         struct ast *y;      /* "true" branch of if statement */
         struct ast *z;      /* "false" branch */
      } flow;
   } u;
};
```

When a new AST node is created it also stores a line number from the configuration file. This value is useful for error reporting of mistakes in the configuration file. The `processed` can be either set to `0` or `1`, if the node was visited during the evaluation of the AST. Each node has its type that denotes the type of an operation, a type of a comparison, a numeric or a string constant, etc. The `value` stores the actual value that the node owns. This value can be either a number or a string. The AST nodes must be connected together, therefore, there is the `x` pointer to the left-side node. Because the definition of the grammar in the configuration file is left recursive, the AST grows to the left side and every AST node must contain the `x` pointer. If this pointer is set to `NULL`, it means, the node is a terminal. These values are common for every AST node.

The union defines specific structures for different node types.

- `node` – this type of the structure covers the vast majority of the AST node types. It contains only a pointer to the right-side branch. Every AST node, that represents some binary operation, utilizes this structure. The left and the right braches are operands for this kind of AST nodes.

- `ref` – this structure is used only in connection with symbols (variables and constants). It includes a pointer to the *symbol table* where each symbol stores its type, a value and an identifier.

- `field` – the structure is specific for the AST nodes that refer to the `get()` and the `getq()` functions. The “`q`” distinguishes between the two functions and the “`f`” is a pointer to a field name.

- `flow` – this structure models the if-then-else statement. The `x` points to the condition part. The `y` and the `z` pointers are "true" and "false" branch respectively.

There are also nodes that does not use the structures in the union such as string literals and numbers.

Since the Bison parses the configuration from the bottom, the parse tree as well as the AST are built from the bottom. Terminal nodes are created at first followed by non-terminals. For instance, if there are lines in the configuration file as follows:

```
var x = "uid: " + get(uid);
exec "echo " + x;
```

the corresesdonding AST tree for this configuration is shown in the picture 5.1. The root of this AST points to the `list` node. This type of the node only connects the other nodes, no operation is defined in it. The following subsection describes how the ASTs are evaluated.

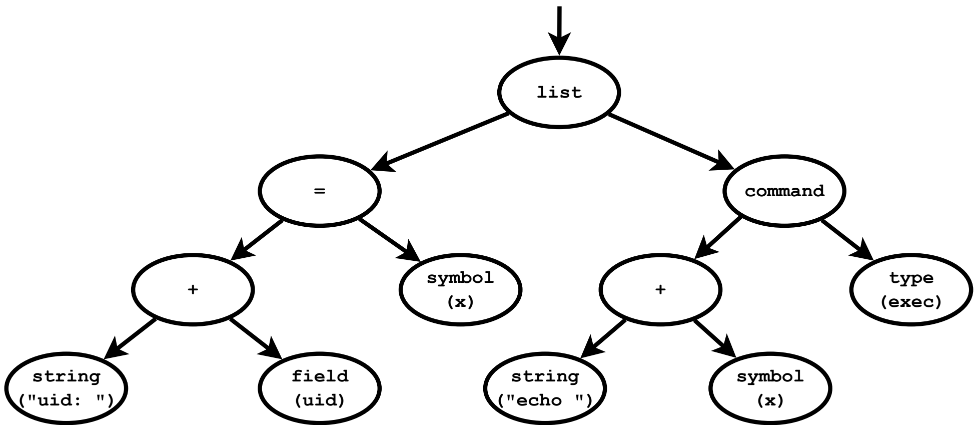


Figure 5.1: Example of the AST.

5.1.2 Evaluation of AST

The evaluation of the AST stands for walking the tree in some way and executing operations defined inside the nodes. The AST nodes can be visited in postorder – the left branch is visited at first, followed by the right branch, and the root is visited at the end. If recursion was used, the function for the AST evaluation could be defined as follows:

```
struct ast_value ast_eval(struct ast *root) {
   struct ast_value v;
   switch(root->type) {
      case AST_OP_ADD:
         v = ast_eval(root->x) + ast_eval(root->u.node.y);
         break;
      case AST_STMT_IF:
          int cond = ast_eval(root->x);
          if (cond)
            v = ast_eval(root->u.flow.y);
          else
            v = ast_eval(root->u.flow.z);
          break;
      ...
    }
    return v;
}
```

The recursion simplifies the definition of the algorithm. The plugin, however, does not employ recursive calling of the `ast_eval()`. This function is not implemented recursively in the plugin, but the principle of the evaluation remains the same.

Because the tree is traversed without using recursion, a stack for keeping the AST nodes is required. The function `ast_eval()` utilizes the other functions for working with the stack and evaluation of the AST nodes:

- `ast_shift()` – this function walks down the tree through the left branch (the `x` pointer) and pushes the visited nodes onto the stack. It does not perform any operations defined in the AST nodes.
- `ast_reduce()` – this function is more complicated than the previous one. It pops the top of the stack (the node is removed from the stack) and takes the current top of the stack (without removing). According to the value of the node from the current top of the stack, there are two possible situations which can come about:
  - the node's value is not defined – it means that the node on the top of the stack have not been assigned any value yet. Because the AST is traversed in postorder, this node waits for the value from the left branch. The popped node represents the left-branch node in this case, thus, its value is assigned to the current top of the stack.
  - the node's value is defined – the value from the left branch have already been assigned to the current top of the stack. Now it is time for the value from the right branch. The popped node represents the AST node of the right branch. There are two values at this point: the first stored in the node on the top of the stack, and the second which is in the popped node. The type of the node on the top of the stack determines what operation should be performed with the two values. The result is then assigned to the node on the top of the stack.

The whole process of the AST evaluation starts with calling of the `ast_eval()`. Inside the function, the `ast_shift()` is called at first. It pushes the nodes onto the stack till the leftmost non-terminal is reached (which is pushed onto the stack too). The `ast_reduce()` considers the popped node to be a terminal node and the current top of the stack to be a non-terminal node. For instance, the pictures 5.2 shows the order in which the AST nodes are evaluated. Every non-terminal visits the left branch at first. This branch must be evaluated to acquire some value which is assigned to the non-terminal. Then the non-terminal waits for some value from the right branch which must be evaluated as well. According to the type of the non-terminal, it is made a decision what operation will be performed with the two values. The result is assigned to the non-terminal. In the next step, this non-terminal will be treated as a terminal. The `list` non-terminal from the piscture 5.2 does not perform any operation. It only connects the AST nodes.

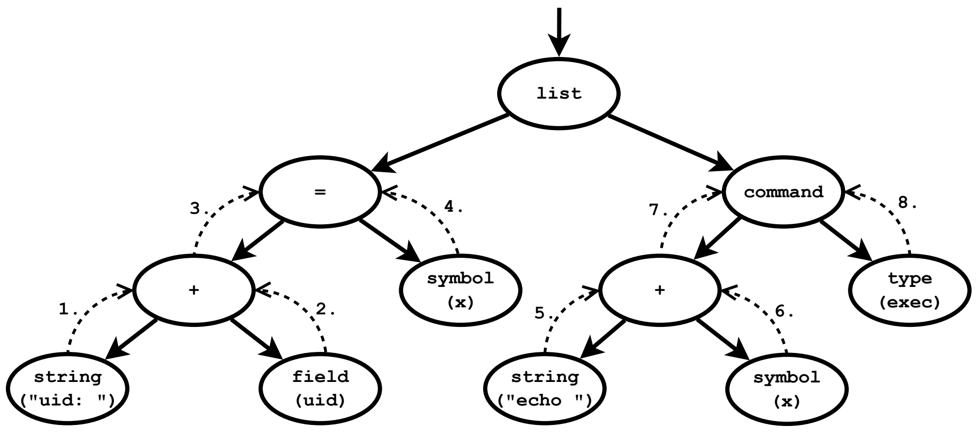


Figure 5.2: Example of the AST evaluation.

The value of every AST node is defined as follows:

```
struct ast_value
{
   enum {
      T_UNDEF, T_NUM, T_STR, T_STR_ALLOC
   } type;

   long n;         /* number */
   char *s;        /* string */
};
```

Terminal values are not allowed to be their `type` set to `T_UNDEF`. This value type would be assigned to the corresponding non-terminal which would not be able to perform any operations. The `T_STR_ALLOC` is the value type that refers to the temporary strings. They might come into the existence as the AST is evaluated, especially during string concatenation. The `struct ast_value` is also a part of the symbol table. Each symbol stores its value in the `struct ast_value` inside the symbol table.

Each time, after the `ast_eval()` is called, the values of the AST must be reset to their original state, so that the AST can be evaluated again. The function that does the job is called `ast_reset()` – it resets non-terminals's `type` to `T_UNDEF` and possibly it frees temporary strings that were allocated during the evaluation. It does not change any terminal values, they must be left unaltered for the future use.

In order to perform the semantic analysis, the whole AST, that represents the plugin's configuration, must be evaluated. This is performed by calling the `ast_eval()` function. One of the function's parameters is the evaluation type. If it is set to the `EVAL_TEST`, the whole tree is evaluated, but no commands (such as `exec`, `add`, `del`, etc.) are executed. If no error occurs during this phase, the semantic analysis is successful. The plugin is now ready to start receiving and checking audit events and trigger the appropriate reactions.

5.1.3 Representation of reactions

Every reaction is divided into two parts:

- **condition** – this part of the reaction is evaluated every time an audit event is received by the plugin
- **action** – this part is only evaluated when the condition part is true

The process of building the whole AST has been discussed so far. The AST includes roots of reactions (the conditions and the actions) that must be accessible for the evaluation. The plugin cannot waste processor time looking for these roots each time an audit event is received. Because of that, there is a *reaction array* for storing the condition and the action pointers of each reaction.

The definition of the grammar contains a certain point where the definition of a reaction is complete, which is depicted in the picture 5.3. The AST node that represents a reaction is a non-terminal. Its `x` pointer is the condition part and the `u.node.y` is the action part of a reaction definition. These pointers are inserted into the reaction array, so that the condition and the action part of each reaction can be accessed quickly.

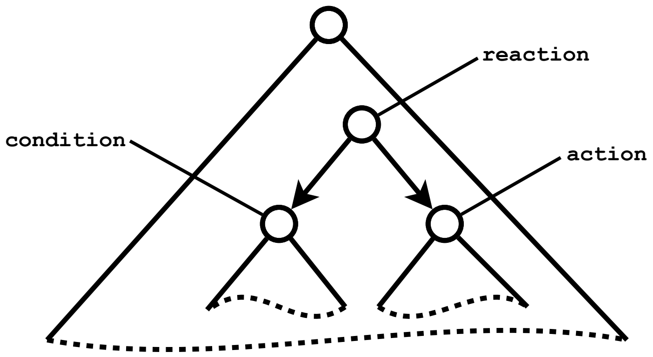


Figure 5.3: Reaction inside the configuration tree.

5.1.4 Execution of commands

Commands `add`, `del`, `addw`, `delw` and `exec` can be placed into the action part of a reaction definition. Each command creates a new process by calling the function `system()`. It utilizes the `fork()` function for creating a new process. The `system()` has one parameter which is a command string that refers to a shell command. Calling the function `system()` is similar to the execution of the command "`/bin/sh -c command`" [15].

Since the `exec` is able to execute any command and the command string can be composed of any data gained from an audit event, this could lead to a security risk. For example, if the `get()` function is used to compose the command string, it could return the value such as "`;rm -rf / ;#`". It would result in erasing of the entire disk. Thus, there must be a way of how to make the value that is returned from an audit event secure for the execution. This is done by calling the `getq()` function (the "`q`" stands for quoted). This function adds quotes at the beginning and at the end of the returned value. Moreover, it replaces quotes within the value with the white space character. The value from the example above would now look like "`';rm -rf / ;#'`" which does not pose any security risk.

Due to the fact that the plugin might be required to handle lots of audit events within a short time period, it must evaluate the condition and the action parts of reaction definitions as fast as possible. As the consequence, the processes that represent the commands are executed in the background. The parent process (the plugin) does not wait for any return codes.

5.1.5 Processing audit events

There is no standard that defines the format of audit events and audit records. Likewise, no standard describes the audit fields that can acquire different field names and field values. As an audit record is received, it is parsed using the `auparse` library. The library can determine the type of some field values according to the field names. The types of the field values for the rest of the field names must be defined by the plugin, which contains its internal list of the field names and the corresponding value types. The field values are accessed by means of the `get()` and the `getq()` functions. These functions are part of expressions where the type of the field value is important. For example, if there is a field value that represents a number and the plugin treats it as a string, then this value lacks a set of operations that could be performed with it. There are more operations that can be performed with numbers than strings.

Every audit event received by the plugin must be processed – its field names and field values must be extracted. The retrieved information is then kept in the data structure where the field values can be accessed by calling the `get()` and the `getq()` functions. This data structure is a hash table – it will be referred as the *field table*. It was chosen due to the fact, that there are no standard field names, and the plugin does not have sufficient information what to expect. If there were standardized audit fields, the field table would not be necessary. The plugin would "know" what field names could appear in audit events. Their corresponding field values would be saved in a different data structure where they could be accessed directly, without hashing the field names.

When the plugin receives an audit event, it parses and stores all the fields from the first record. The first record involves the most important information. The rest of the records only refer to auxiliary data. These records are not processed apart from the the `CWD` and `PATH` records (if they are present). The two records include the fields that can be used to construct an absolute path. This value is related to some event that refers to some object in a file system. There are basically two situations that can occur when getting the absolute path:

- the `name` field in the `PATH` record starts with the "`/`" – the `name` field already contains the absolute path.
- the `name` field in the `PATH` record starts with the "`.`" – the `name` field contains only a relative path. This field must be concatenated with `cwd` field from the `CWD` record to form the absolute path.

There might also be more `PATH` records with various values of the `name` field. The plugin selects the longest value of the `name` field from the `PATH` records. This value is the most likely to be the absolute path. The absolute path is then put into the field table as well. It can be looked up using the `apath` field name.

The first audit record might contain the same field name more times (but with various values). This is the case of the `key` field. In order to keep all the field values, the field table implements a list of values instead of a single value per one field name. Each value list includes the `which` pointer that selects one of the values from the value list as is shown in the picture 5.4.

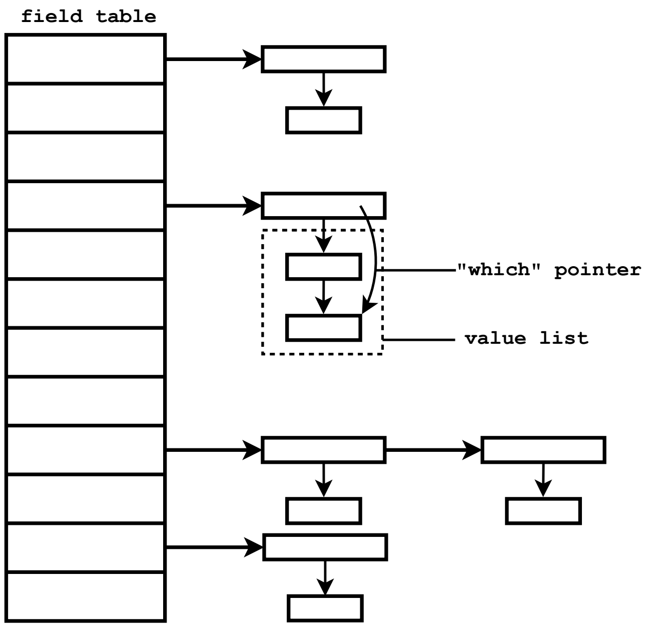


Figure 5.4: Value list in the field table.

The mulptiple-value field can be a part of:

- the condition part – the whole condition tree is evaluated for each value individually till the condition is true or the evaluation was performed with all the values. The `which` pointer always selects the value being in use.
- the action part – only the value selected with the `which` pointer is used for the evaluation.

The plugin supports only the `key` field to contain more values, which is sufficient. When the condition is true, the `which` pointer is set to the value that caused the condition to be true. This value is also used in the action part. If the multiple-value field is only included in the action part of a reaction, the `which` pointer chooses the last value in the audit record.

5.1.6 Plugin's components

The previous subsections dealt with implementation details mainly. On the other side, this subsection shows the overall design of the reactive plugin. The plugin is divided into several components that are joined together in different ways.

The main components of the `audisp-reactive` are:

- **audit event processor**
- **field table**
- **reaction array**
- **symbol table**
- **AST evaluator**

The audit event processor extracts audit field names and field values from audit events. These values are put into the field table where they can be read by calling the `get()` and the `getq()` functions. The reaction array stores pointers to the condition and the action part of each reaction. Since the AST includes symbols (variables and constants), there is the symbol table where they are stored. The last component, the AST evaluator, is able to evaluate ASTs – it takes pointers to the ASTs from the reaction array. The components and their relationships are depicted in the picture 5.5.

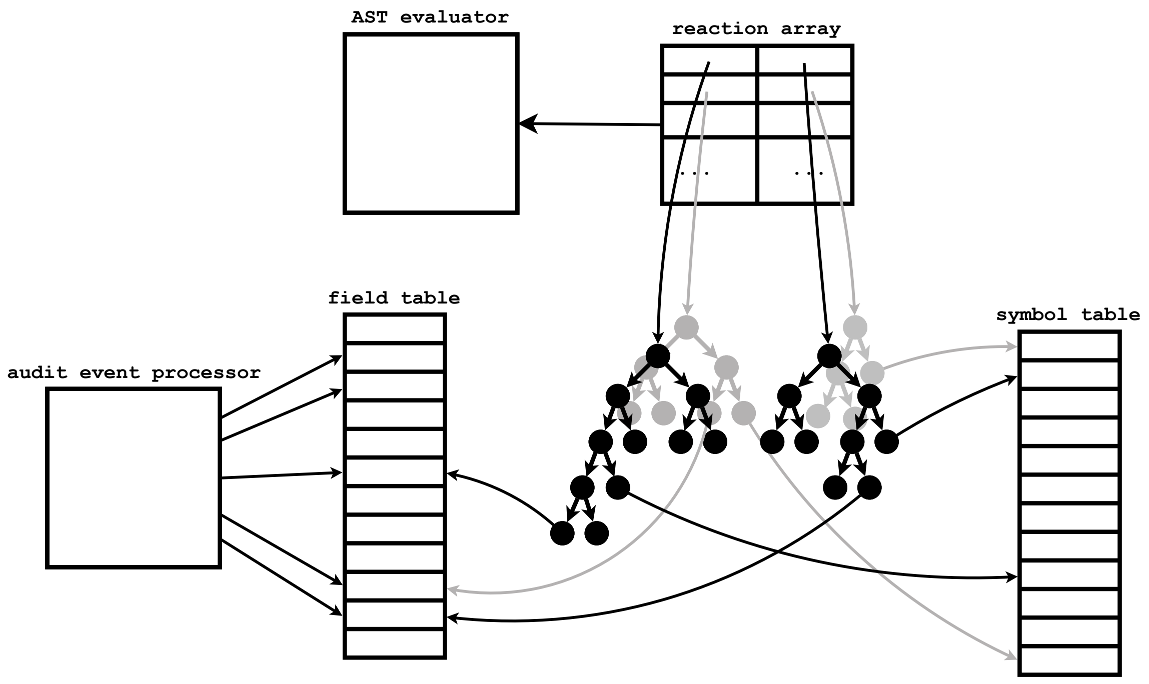


Figure 5.5: Components of the plugin.

5.1.7 Trigerring reactions

When the configuration is read and the AST is created, the plugin waits for audit events to be analyzed. The following pseudocode explains what is being performed by the plugin step by step from the point when a new audit event is received to the point of setting off a reaction. The part of the plugin which receives audit events is not included in the pseudocode. It consists of an endless loop where a file descriptor is checked for changes indicating that the I/O operation of reading an audit event can be performed.

Algorithm 1 Processing an audit event by the reactive plugin

```
 1: new audit event arrived
 2: for all records in audit event do
 3:   if first record then
 4:     for all fields in the first audit record do
 5:       get field name
 6:       get field value
 7:       insert field name and field value into field table
 8:     end for
 9:   else if CWD record then
10:     get and save value of cwd field
11:   else if PATH record then
12:     get the longest value of name field
13:   end if
14: end for
15: if PATH record is present then
16:   construct absolute path
17:   insert apath field name and absolute path into field table
18: end if
19: for all reactions from reaction array do
20:   evaluate condition
21:   while condition should be evaluated again with the next value do
22:     reset condition
23:     set the next value
24:     evaluate condition
25:     if condition is true then
26:       stop while
27:     end if
28:   end while
29:   if condition is true then
30:     evaluate action
31:     reset action
32:   end if
33:   reset condition
34: end for
35: clear field table
```

For example, the rule database in the kernel is changed by adding the audit rule as follows:

```
auditctl -a exit,always -F path=/etc/shadow -F success=0 -S open -k k1 -k k2
```

It means, that the kernel generates the audit event if there in an unsuccessful attempt to open the file `/etc/shadow`. The generated audit event will also contain the two keys: “`k1`” and “`k2`”. The example of such audit event is as follows:

```
type=SYSCALL msg=audit(1274629969.319:42): arch=40000003 syscall=5
success=no exit=-13 a0=bfea38da a1=8000 a2=0 a3=bfea305c items=1
ppid=1620 pid=1649 auid=500 uid=500 gid=500 euid=500 suid=500 fsuid=500
egid=500 sgid=500 fsgid=500 tty=pts1 ses=4 comm="cat" exe="/bin/cat"
```

```
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=6B31016B32
type=CWD msg=audit(1274629969.319:42):  cwd="/home/user"
type=PATH msg=audit(1274629969.319:42): item=0 name="/etc/shadow"
inode=88055 dev=fd:00 mode=0100000 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:shadow_t:s0
```

In order to react to the above audit event, the configuration file must define the reaction whose condition part will be evaluated as "true". The following reaction definition meets this requirement:

```
react: get(key) == "k2" && get(success) == "no" &&
       get(apath) == "/etc/shadow"
{
   add "exit,always -F uid=" + get(uid) + "-F success=0 -S open";
}
```

The lines from 2 to 18 in the pseudocode refer to parsing and extraction of the field names and field values from the audit event. This phase involves the audit event processor and the field table – the former one retrieves the information which is put into the field table. As regards the absolute path, it is contained in the `PATH` record. The `CWD` record is not needed to construct the absolute path in this case.

After the audit event processor finishes its job and the field table is filled, the plugin starts checking all the condition parts of the reactions one by one. It is performed by reading the condition pointers from the reaction array and passing them to the AST evaluator to be evaluated. There is just one reaction definition in the example so that only one condition is evaluated. As soon as the current condition is true, the corresponding action is set off immediately. The whole process is described between the lines 19 and 34. The action from the example involves adding a new audit rule to the kernel. According to the example, if some user tries to open the `/etc/shadow` file, the plugin will add the new audit rule for monitoring this user.

There is also the multiple-value field in the example – using the `get(key)` function in the condition, the values "`k1`" and "`k2`" can be accessed. The evaluation of this kind of field is described between the lines 19 and 26 in the pseudocode. After every evaluation is done, the AST representing either a condition or a action, must be reset (the lines 22, 31 and 33). Likewise, the field table must be cleared (line 35) which prepares this structure for the upcoming audit event.

5.2 Audisp-stats

The plugin provides some statistic information that the `audisp-reactive` can take advantage of. The information can be used for making decisions inside reaction definitions.

5.2.1 SQLite

Because the plugin is supposed to provide statistic information, it utilizes a database for storing the audit data. In spite of the fact that the same information can be found in the log file, the database must be used. It must be possible to search certain audit data using complicated queries as fast as possible and the database is much faster in this way.

Moreover, the statistic information is supposed to be retrieved only from the audit events that are of interest. The log file stores all the audit events, but the database stores only those interesting ones.

The plugin stores the audit data by means of SQLite – the database is represented by a database file and no additional process, such as the database daemon is not required to work with the stored data. Compared to MySQL, for example, a daemon is needed to work with the database which is not acceptable for the audit. As the system boots, the audit is started before the other daemons, including the MySQL daemon. It could result in loss of audit data generated between the start of the audit daemon and the start of the database daemon. This is the main reason why the SQLite was chosen to store the audit data by this plugin.

5.2.2 Database schema

There are three tables that form the database schema, as is shown in the picture 5.6. When inserting audit data into the database, not the whole audit events are stored. The configuration file defines what field names are of interest. The plugin stores only the field values that refer to these field names. It is reflected by the `field` table which can be composed of a variable number of columns. Every field name defined in the configuration refers to one column in the field table. For example, there are `uid` and `gid` in the configuration so that there will be `uid` and `gid` columns in the `field` table.

Because one record can include one or more `key` fields, this fact is expressed by the `keyfield` table. It stores only keys for the associated audit event. The plugin is assumed to provide time-related statistics, therefore, timestamps of audit events must be stored as well. The time information could be a part of the `field` table, however, it would result in storing redundant data. If there were two or more identical audit records differing only in their timestamps, the same information would be saved more times. Because of that, there is another table, called `timestamp`, that keeps only time information and prevents from storing redundant data.

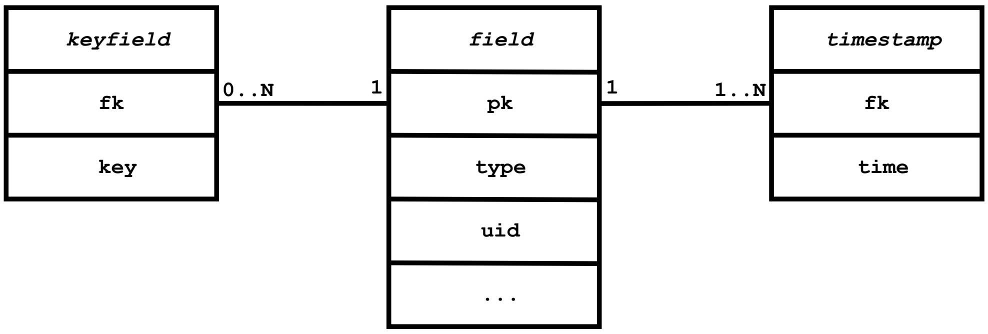


Figure 5.6: Database schema.

Before the database tables are created, the plugin's configuration file is read. Each line in the configuration file consists of a field name and its type (either string or integer). The type of a field name is defined explicitly as there is no standard that determines what field names are associated with what value types.

The sql queries for creating the tables are the following:

```
CREATE TABLE field (
   pk INTEGER PRIMARY KEY,
   <field_name> <field_type>
);

CREATE TABLE keyfield (
   fk INTEGER NOT NULL,
   key VARCHAR
);

CREATE_TABLE timestamp (
   fk INTEGER NOT NULL,
   time DATE
);
```

The first query is built based on the field names and field types from the configuration file. For example, if there is the configuration as follows:

```
type = string
uid = int
gid = int
key = string
exe = string
```

the plugin substitutes the `<field_name> <field_type>` pairs with:

```
type VARCHAR, uid INTEGER, gid INTEGER, exe VARCHAR
```

The length of `VARCHAR` does not have to be specified, the SQLite is capable of storing strings of any length. In the example above, the `key` field is skipped, because this field is stored in its own table. In case, there is no `key` field in the configuration file, the `keyfield` table is created anyway. It must be present due to the other queries for inserting and searching that expect the `keyfield` table to exist. The `timestamp` table is always created regardless to the plugin's configuration.

The `field` table defines the column `pk` as "`INTEGER PRIMARY KEY`" which according to [18] will autoincrement. If the `NULL` is inserted into this column, the `NULL` is automatically converted into an integer which is unique over all primary keys currently in the table. It might overlap with the keys that have been previously deleted from the table.

5.2.3 Storing information from audit events

Before any data is passed to the database to be stored, the plugin checks for the "`stats`" key in every received audit event. If the key is present, it indicates that the audit event should be saved. Although, some audit events cannot contain the key field. This applies for the audit events that are generated by the trusted applications such as Pam, sshd, gdm, etc. Therefore, the configuration file of the plugin can also define that these events should be put into the database. Before an audit event can be stored, it must be parsed to retrieve field values. The `audisp-stats` is similar to the `audisp-reactive` in how an audit event is processed – the first record as well as the records that contain the absolute path are parsed and the fields are put into the hash table (see the subssection 5.1.5). The extracted field values are used to build SQL queries for inserting and searching data.

New information is inserted into the database in several steps using various SQL queries. Some queries are generated from the field values kept in the hash table. In order not to access the hash table more times to get the same field value, there is the array of pointers that is filled each time a new audit event is received. The array is used for quick access to the field values stored in the hash table. These pointers associate field names from the configuration with field values from the hash table. If there is not such field name in the hash table that the configuration defines, it is reflected by setting the pointer to the `NULL`. The array has only pointers to interesting fields defined in the configuration file.

First of all, field values must be searched to figure out whether they are already saved or not. This does not apply for the timestamps. The SQL query for searching is as follows:

```
SELECT pk FROM field
WHERE <condition> AND
(SELECT COUNT(*) FROM keyfield WHERE fk = pk AND
key IN (<keys>)) = <key_count>;
```

The `<condition>` part is constructed according to the field names in the configuration file and field values in the hash table. The field names are concatenated either with the "`=`" and the corresponding field value, or with the "`IS NULL`". It depends on whether the field name is present in the hash table or not. The values of the `key` fields must be checked as well. The `<keys>` is substituted with these values and the `<key_count>` stands for their count. An audit event might not contain all the field names defined in the configuration file so that the "`NULL`" values are inserted into the `field` table. They must be matched as well to make a decision whether the audit event is already saved.

In case the previous query does not return the primary key of an audit event, it means the audit event with the specified values have not been inserted to the table yet. Therefore, it will be stored. For this purpose, the SQL query for inserting values is needed – it is generated from the field values in the hash table as follows:

```
INSERT INTO field VALUES (NULL, <field_values>);
```

The `<field_values>` stands for the string of field values separated with a comma – the number of field values equals to the number of the columns the `field` table has (without the primary key's column). Providing that, the configuration defines the `key` field to be stored, this operation is performed now. The SQL query that is used for this purpose needs to be generated, too:

```
INSERT INTO keyfield VALUES(<field_primary_key>, <key>);
```

The `<field_primary_key>` is the primary key from the `field` table and the `<key>` is the value of the `key` field. This query is executed as many times as there are the `key` values.

If the audit event is already in the table, the selection query returns the primary key of the audit event. The `field` and `keyfield` tables are not changed in this case. Regardless to the result of the selection query, however, the time information must be inserted into the `timestamp` table. This is performed using the query:

```
INSERT INTO timestamp VALUES (<field_primary_key>, <timestamp>);
```

The `<field_primary_key>` is the primary key of the audit event from the `field` table and the `<timestamp>` refers to the audit event's timestamp.

The whole process of storing an audit data is described by the following pseudocode:

Algorithm 2 Storing audit event by the statistic plugin

```
 1: new audit event arrived
 2: if audit event should be stored then
 3:    save audit data into field table
 4: else
 5:    do not store audit event
 6: end if
 7: for all field names in configuration do
 8:    get pointer to field value in field table and store it into array
 9: end for
10: generate and execute selection query
11: if selection query does not return any row then
12:    generate and execute insertion query for „field“ table
13:    if field values referring to „key“ field name should be saved then
14:       for all field values referring to „key“ field name do
15:          generate and execute insertion query for „keyfield“ table
16:       end for
17:    end if
18: end if
19: generate and execute insertion query for „timestamp“ table
20: clear field table
```

The third line represents the lines 2 - 18 in the pseudocode from the subsection 5.1.7. In case the received audit event should be saved, it is parsed to retrieve audit data (line 3). The array that associates the field names of interest with their field values in the hash table is set between the lines 7 and 9. The core of the insertion of new audit data is located between the lines 10 and 19. Because one insertion can be composed of several SQL insert queries, they must be executed atomically to keep database consistent. That is why these queries are part of a transaction. If an error occurs while inserting data into the tables, the whole insertion is rolled back to the state before the insertion.

5.2.4 Plugin's components

This plugin can be split into a couple of components:

- **audit event processor**
- **hash table**

- **field array**
- **associative array**
- **SQL unit**
- **database file**

The hash table is the same component as the field table from the `audisp-reactive`. It is called the hash table here since there is the field table in the database. The field array only contains field names and their value types as defined in the configuration. The association array id the array of pointer for quick access to the hash tables. Another component, the SQL unit generates and performs execution of SQL queries over the database file. These components are depicted in the picture 5.7.

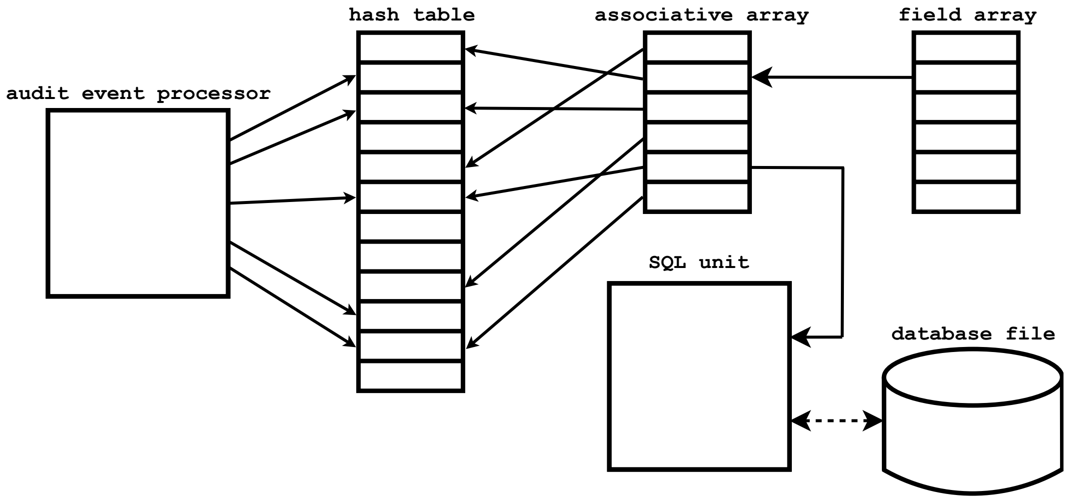


Figure 5.7: Components of the plugin.

5.2.5 Statistics

The plugin is able to provide time-related statistics about how many times something happened within a time period. The `field` and the `keyfield` tables store information that are used for looking up a certain event, while the `timestamp` table keeps timestamps of the events. The time information is stored in the form of the Unix time. It is represented as an integer number. The SQL query that returns the occurance of some specified audit events within a certain time period is the following:

```
SELECT COUNT(*) FROM
(SELECT DISTINCT pk FROM field LEFT JOIN keyfield
ON field.pk = keyfield.fk
WHERE <condition>) AS data
JOIN timestamp ON data.pk = timestamp.fk
WHERE time >= <start_point> AND time <= <end_point>;
```

The `<condition>` part defines what field names and field values are the interesting ones, the `<start_point>` is the time in form of the unix time It refers to the start poin of a time interval and the `<stop_point>` is the end of the time interval.

The database might store a large number of events, therefore, the old data should be erased periodically. This can be done by means of the `cron` daemon that executes the SQL queries for deleting old records from the database:

```
DELETE FROM timestamp WHERE time <= <time>;
DELETE FROM field WHERE pk NOT IN (SELECT fk FROM timestamp);
DELETE FROM keyfield WHERE fk NOT IN (SELECT pk FROM field);
```

The `<time>` is used to determine what records are old and should be removed. These SQL queries are also enclosed in a transaction to keep the data consistent.

5.2.6 Communication with audisp-reactive

The `audisp-reactive` is assumed to utilize the `stats()` function within its configuration file. However, the syntax of the configuration file must add support for the function and its parameters. The first parameter is a part of the `WHERE` clause in the selection SQL query from the previous subsection. It can be built using string concatenation. Thus, the resulting string can be also composed using the `get()` and the `getq()` functions. The next two parameters are start and stop points in time that form a time interval. They might be expressed using “`now`”, “`sec`”, “`min`”, etc. keywords. The example of calling the `stats()` function might be as follows:

```
var x = stats("uid=" + get(uid) + " AND key=" + getq(key), 2 hour, now);
```

The `audisp-reactive` must represent this function in the AST, therefore, there were added two AST nodes (`stats` and `period`) as shown in the picture 5.8. The `expression` node

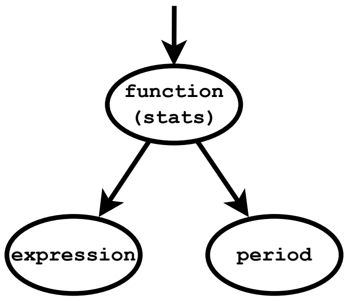


Figure 5.8: New AST nodes.

stands for some expression that forms the SQL query. The tree is evaluated just like any other AST – the left branch is visited at first, its value is assigned to the non-terminal that refers to the calling of the `stats()` function. Then, the right branch is visited, where a time interval is stored. Finally, the non-terminal is evaluated by calling the function provided by the `audisp-stats`, which has the following prototype:

```
long sql_get_stats(const char *cond, const unsigned int start,
                   const unsigned int stop);
```

The `struct ast` had to be extended with the new type of AST node that represents the time interval:

```
struct ast
{
   ...
   union {
      ...
      struct {
         unsigned int start;
         unsigned int stop;
      } period;
   } u;
};
```

Chapter 6

Use of plugins

This chapter describes different examples in which the two plugins, the `audisp-reactive` and the `audisp-stats`, take part.

6.1 Reflecting system state by adding and deleting audit rules

The audit is able to monitor the system according to the rules in the rule database. These rules are put into to kernel by a system administrator, who responsible for adding or deleting some of them when something changes. For example, a new disk partition that should be audited for file operations is mounted. Because of that, the system administrator must add the appropriate audit rules to the rule database using the `auditctl`. Although, the reactive audit makes it possible to add these rules automatically, as soon as the partition is added without any assistance of a system administrator. The first example shows how to deal with this kind of situation using the `audisp-reactive`.

6.1.1 USB stick

This example demonstrates how new rules are added (deleted) when a USB stick is attached (detached). These rules can monitor what files are transferred to or from the USB stick. The audit events generated using these rules can be used for finding out if there was some important data stolen from a computer.

Since the `audisp-reactive` is supposed to react to mount and umount audit events, the kernel must generate them. In order to generate these events, the kernel must contain the appropriate audit rules. They are added by means of the `auditctl` as follows:

```
auditctl -a exit,always -S mount
auditctl -a exit,always -S umount
```

The `-S` parameter specifies what system calls should be audited. After these rules were added, they can be checked by the `auditctl -l` command, that shows the content of the rule database in the kernel:

```
LIST_RULES: exit,always syscall=mount
LIST_RULES: exit,always syscall=umount
```

The configuration file of the `audisp-reactive` must contain the reaction definition whose condition part will be evaluated as “true” when the mount (umount) audit event is generated.

Before defining such reaction, the audit event must be examined to determined what field names and field values will be included in the reaction's condition. The first audit event was generated when the USB stick was attached:

```
type=SYSCALL msg=audit(1274393892.929:47): arch=40000003 syscall=21
success=yes exit=0 a0=2ea41d8 a1=2ea41e8 a2=2ea4200 a3=c0ed0006 items=2
ppid=1453 pid=1833 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=(none) ses=4294967295 comm="mount" exe="/bin/mount"
subj=system_u:system_r:mount_t:s0-s0:c0.c1023 key=(null)
type=CWD msg=audit(1274393892.929:47):  cwd="/"
type=PATH msg=audit(1274393892.929:47): item=0 name="/media/flash"
inode=91993 dev=fd:00 mode=040700 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:mnt_t:s0
type=PATH msg=audit(1274393892.929:47): item=1 name=(null) inode=19787
dev=00:10 mode=060660 ouid=0 ogid=6 rdev=08:11
obj=system_u:object_r:fixed_disk_device_t:s0
```

When the USB stick was detached, the following audit event was created:

```
type=SYSCALL msg=audit(1274394047.953:48): arch=40000003 syscall=22
success=yes exit=0 a0=239f5c0 a1=bfc8ef60 a2=fc6184 a3=239f5c1 items=1
ppid=1453 pid=1842 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=(none) ses=4294967295 comm="umount" exe="/bin/umount"
subj=system_u:system_r:mount_t:s0-s0:c0.c1023 key=(null)
type=CWD msg=audit(1274394047.953:48):  cwd="/"
type=PATH msg=audit(1274394047.953:48): item=0 name="/media/flash" inode=1
dev=08:11 mode=040700 ouid=500 ogid=500 rdev=00:00
obj=system_u:object_r:dosfs_t:s0
```

The audit fields which are in the boxes are interesting for the plugin's configuration. The `syscall` and the `success` fields are used to recognize the right mount and the umount audit events that will be reacted to. The `name` field can be used in the action part of the reaction. The configuration file includes two reaction definitions:

```
  react: get(syscall) == 21 && success == "yes" {
     addw get(apath) + " -p w";
  }

  react: get(syscall) == 22 && success == "yes" {
     delw get(apath) + " -p w";
  }
```

The first reaction adds a new audit rule to the kernel. It is constructed from the `name` field that represents the absolute path of the mount point. The second one removes the audit rule which was added using the first reaction. Now that everything is set, the USB stick can be attached to any mount point. As soon as it happens, the reactive plugin starts to monitor what is sent to the USB stick.

The example requires two audit rules to be in the kernel permanently. However, the number of the permanently stored audit rules can be reduced to just one. It is achieved by changing the plugin's configuration. The only rule that is needed to be permanently stored in the kernel is the one used for generating mount events:

```
auditctl -a exit,always -S mount
```

The audit rule for the generation of the umount audit events is placed inside the definition of the first reaction:

```
react: get(syscall) == 21 && success == "yes" {
   add "exit,always -S umount";
   addw get(apath) + " -p w";
}

react: get(syscall) == 22 && success == "yes" {
   delw get(apath) + " -p w";
}
```

The second reaction definition is unchanged. This example works exactly as the last one. The only difference is in the number of permanently stored audit rules in the kernel. It is better to keep less rules in the kernel. The more audit rules being in use, the more processor time it takes to match them with events originating in a system.

6.1.2 Different users

The second example is more complicated than the previous one. It is assumed, that there are various users and a system administrator needs to monitor activities specific for each one. The `audisp-reactive` must be able to add a set of rules for a specific user when the user logs in. Furthermore, it is required that the set of audit rules will be removed as the user logs out. It is analogous to the example with the USB stick, although, the same user can be logged in more times. The audit events related to the user logins and logouts are generated without using any audit rules in the kernel. There is no need using the `auditctl` for adding any permanent audit rules in the kernel. The user's identity can be recognized using the audit event that contain the `USER_START` audit record. It is always generated amog the other audit events when a user logs in over the SSH, locally or just uses the `su` command for changing the identity. The example of such audit event (it contains just one audit record) is as follows:

```
type=USER_START msg=audit(1274435897.539:68): user pid=1816 uid=0 auid=0
ses=5 subj=unconfined_u:system_r:sshd_t:s0-s0:c0.c1023
msg='op=PAM:session_open acct="root" exe="/usr/sbin/sshd" hostname=10.0.2.2
addr=10.0.2.2 terminal=ssh res=success'
```

On the other side, when the user logs out, the similar audit event is generated. It is composed of one audit record as well:

```
type=USER_END msg=audit(1274435957.494:72): user pid=1816 uid=0 auid=0
ses=5 subj=unconfined_u:system_r:sshd_t:s0-s0:c0.c1023
msg='op=PAM:session_close acct="root" exe="/usr/sbin/sshd" hostname=10.0.2.2
addr=10.0.2.2 terminal=ssh res=success'
```

The `type` and `res` fields are checked by the `audisp-reactive` so that successful logins and logouts are recognized. The `acct` field can be contained in the action part of the reaction definition. Moreover, the plugin's configuration must pay attention to how many times a certain user is logged in. There are global variables that are incremented with every login and decremented as users log out. The plugin must keep this information in order not to add the same audit rules with each login. Furthermore, if a certain user is logged in more times, the set of the audit rules specific for this user cannot be deleted with every single logout. The rules ought to be deleted when the user is not present in the system. It means, that the user must log out as many times as he/she was logged in. The reaction definitions that model this situation are the following:

```
var login_cnt_user1 = 0;
var login_cnt_user2 = 0;
# Login
react: get(type) == "USER_START" && get(res) == "success" {
   var acct = get(acct);
   if (acct == "user1") {
      if (login_cnt_user1 == 0) {
         # add specific rules for user1
      }
      login_cnt_user1 = login_cnt_user1 + 1;
   } else if (acct == "user2") {
      if (login_cnt_user2 == 0) {
         # add specific rules for user2
      }
      login_cnt_user2 = login_cnt_user2 + 1;
   }
}
# Logout
react: get(type) == "USER_END" && get(res) == "success" {
   var acct = get(acct);
   if (acct == "user1") {
      login_cnt_user1 = login_cnt_user1 - 1;
      if (login_cnt_user1 == 0) {
         # delete specific rules for user1
      }
   } else if (acct == "user2") {
      login_cnt_user2 = login_cnt_user2 - 1;
      if (login_cnt_user2 == 0) {
         # delete specific rules for user2
      }
   }
}
```

These reactions reduce the number of audit rules being in use, therefore, the kernel checks events occurring in the system faster.

6.2 Sending warnings to administrator

Since the reactive plugin can check various audit events, it is also able to reveal if something suspicious is going on. Moreover, the plugin can execute any commands by means of the `exec` command. Thus, it is possible to send warnings to a system administrator in case there are some unusual audit events.

6.2.1 File access

For example, changing the files in the `/etc` by writing to them is considered to be suspicious, especially when this operation is not performed by a system administrator. There must be the audit rule in the kernel that is used to generate audit events about such activities:

```
auditctl -w /etc -p wa -k warning
```

The audit event that was generated when an unprivileged user tried to change the file `/etc/passwd` is as follows:

```
type=SYSCALL msg=audit(1274458207.148:116): arch=40000003 syscall=5
success=no exit=-13 a0=834e870 a1=8241 a2=1b6 a3=0 items=1 ppid=1750
pid=1798 auid=500 uid=500 gid=500 euid=500 suid=500 fsuid=500 egid=500
sgid=500 fsgid=500 tty=pts3 ses=12 comm="nano" exe="/bin/nano"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="warning"
type=CWD msg=audit(1274458207.148:116):  cwd="/home/user"
type=PATH msg=audit(1274458207.148:116): item=0 name="/etc/passwd"
inode=88267 dev=fd:00 mode=0100644 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:etc_t:s0
```

The `key` field is used to distinquish this kind of audit events by the `audisp-reactive`. The other fields, which are in boxes, are used to create a brief message for a system administrator. The definition of the reaction might be as follows:

```
react: get(key) == "warning" {
   var msg = "Warning(File access): path: " + get(apath) + " success: " +
             get(success) + " command: " + get(exe);
   exec # command for sending e-mail or other type of message
}
```

6.2.2 SSH server

Provided that unsuccessful authentication attempts via SSH should be controlled and reacted to if this number is high, the reactive audit can be configured for this scenario as well. This example shows how the two plugins, the `audisp-reactive` and the `audisp-stats`, cooperate. The audit events about authentication attempts are generated without using any rules in the kernel. The example of such audit event, which includes only one audit record of type `USER_LOGIN`, is as follows:

```
type=USER_LOGIN msg=audit(1274521811.039:3445): user pid=1830 uid=0
auid=4294967295 ses=4294967295 subj=system_u:system_r:sshd_t:s0-s0:c0.c1023
msg='acct="user": exe="/usr/sbin/sshd" hostname=? addr=10.0.2.2
terminal=sshd res=failed'
```

In order not to react to every single unsuccessful login attempt, the `audisp-stats` stores the interesting fields. They can be used to retrieve the number of unsuccessful login attempts within a time period. Because of that the configuration file of the `audisp-stats` must store at least the fields: `type`, `acct`, `exe` and `res`. The timestamp of each event is stored as well. Now that the `audisp-stats` is set correctly, the configuration of the `audisp-reactive` can be defined:

```
var msg_sent = 0;

react: get(type) == "USER_LOGIN" && get(exe) == "/usr/sbin/sshd" &&
       get(res) == "failed" {
   var query = "type='USER_LOGIN' AND exe='/usr/sbin/sshd' AND " +
               "res='failed'";
   var login_cnt = stats(query, 1 day, now);

   if (login_cnt > 50) {
      if (!msg_sent) {
         var msg = "Warning(Login attempts): count: " + login_cnt;
         exec # command for sending e-mail or other type of message
         msg_sent = 1;
      }
   } else if (login_cnt < 20) {
      if (msg_sent)
         msg_sent = 0;
   }
}
```

Because the plugins represent two independent processes, they are not synchronized. Therefore, it is not possible to use "`login_cnt == 50`". For example, an audit event is received and processed by the both plugins at the same time. The `audisp-reactive` retrieves some statistic information (number 49 is returned), then the `audisp-stats` inserts new audit data into the database. When the next audit event is received, the plugins may process it in the opposite order – the new audit data is put into the database at first, then the `audisp-reactive` retrieves the statistic information which will be the number 51. The number 50 is skipped.

Using the `audisp-stats`, it is possible to monitor unsuccessful login attempts of a certain user, too. In order to do that, the `query` from the example above can be composed using the `getq()` function:

```
"type='USER_LOGIN' AND exe='/usr/sbin/sshd' AND res='failed' AND " +
"acct=" + getq(acct);
```

Furthermore, the `audisp-stats` makes it possible to provide statistic information about the same event within different time intervals. Using this approach, some anomalies can be revealed. The configuration file of the `audisp-reactive` retrieves the statistic information twice in this case:

```
var msg_sent = 0;

react: get(type) == "USER_LOGIN" && get(exe) == "/usr/sbin/sshd" &&
       get(res) == "failed" {
   var query = "type='USER_LOGIN' AND exe='/usr/sbin/sshd' AND " +
               "res='failed'";
   var login_cnt_current = stats(query, 1 day, now);
   var login_cnt_old = stats(query, 2 day, 1 day);
   var difference = login_cnt_current - login_cnt_old;
```

```
    if (difference > 30) {
       if (!msg_sent) {
          var msg = "Warning(Login attempts - anomaly)";
          exec # command for sending e-mail or other type of message
          msg_sent = 1;
       }
    } else if (difference < 15) {
       if (msg_sent)
          msg_sent = 0;
    }
}
```

6.3 Automatic system protection

The previous section discussed sending warnings to a system administrator if something suspicious was going on in a system. The reactive audit does not necessarily send only warnings, it can set off reactions that will protect the system. They are executed by means of the `exec` command. For instance, if a user is trying to modify a certain file without a permission to do so, the `audisp-reactive` will trigger a reaction that will log out the user. The line in the `audisp-reactive` configuration that does the job is the following:

```
exec "pkill -u " + get(uid);
```

Furthermore, the reactive plugin can change configuration of other programs that are in use. In case, there are plenty of unsuccessful login attempts from a certain IP address, the `audisp-reactive` is able to block it. The IP address is included in the corresponding audit event. The command for blocking the IP will add a new rule to the `iptables` firewall as follows:

```
exec "iptables -A INPUT -s " + get(addr) + " -j DROP";
```

There are plenty of other examples that describe how the system could be protected. The `exec` command can be used in many different ways which makes the mechanism of the reactive audit flexible.

Chapter 7

Summary of results

To begin with, the mechanism of the reactive audit as it was proposed in the Chapter 4 was implemented successfully. No changes to the existing user space audit tools had to be done to support the reactive audit. The whole mechanism is implemented within the audit plugin which is called `audisp-reactive`. It receives audit events, extracts audit data from them and makes decisions whether to trigger reactions. This plugin sets off reaction according to the information from the audit events. Furthermore, there was implemented another plugin. It is called `audisp-stats` and its task is to collect audit data from some of audit events that are of interest (such as login attempts, unprivileged file access, etc.). Moreover, it is possible to retrieve time-related statistical information from the saved audit events. The `audisp-reactive` can take advantage of this information for triggering reactions and detection of anomalies.

The `audisp-reactive` makes it possible to keep only necessary audit rules in the kernel. If the system call auditing is on, the kernel checks every system call as it leaves the kernel. The checking stands for matching the system call relevant data (that were collected when the system call visited different parts of the kernel) with the audit rules. The system performance may be improved when less audit rules are in the kernel since there will be less comparisons.

The previous chapter describes a couple of different scenarios where the two plugins are used. To sum it up, the first example shows how the `audisp-reactive` can watch for audit events that are generated when a USB stick is attached or detached. It reacts by adding the audit rule that monitors what is transferred to the USB stick. When the USB stick is detached, the audit rule is removed. The next example shows how the plugin adds and deletes user specific audit rules when the users log in and log out. This useful for reduction of the audit rules in the kernel. Not only changes of the rule database, but also execution of different commands can be performed. This is described in the example where a user tries to write to a file without a permission to do so. The reactive plugin reacts to this event by sending a warning to an administrator. The `audisp-stats` is used in the next example. It provides time-related statistics about failed login attempts. The `audisp-reactive` triggers the reaction of sending a warning to an administrator when the number of unsuccessful logins reaches some number. Additionally, the plugin can send the warning when an anomaly is detected. It means that the number of unsuccessful logins in some time period is much higher than it was in the past. Finally, the last example shows how the system can protect itself by executing commands that will ban users or block IP addresses.

As the examples show, the mechanism of the reactive audit was used in various situations. Additionally, there may be other scenarios in which the reactive is useful. It because this mechanism is general enough to cover a large number of different situations.

Chapter 8

Conclusion

The first chapters of the thesis discuss general principles and requirements of the operating system auditing. Besides, there is given a description of the audit system in Linux, which involves a closer look at the implementation and the overall design as well. According to the information from these chapters, the thesis comes up with the proposal of an extension for the current audit mechanism – the reactive audit. This part explains what the proposed mechanism should be capable of, which affects its implementation.

The next part of the thesis deals with the implementation of the reactive audit. It is implemented entirely within an audit plugin, which means that no changes had to be done to the existing code to support this mechanism. Moreover, the plugin does not necessarily be in use, it can be turned off when auditing. The plugin's configuration file defines what reaction should be set off when a specific audit event is received. In order to recognize the audit events, the plugin must extract information from them by parsing. Although, there is no standard describing the format of the audit events. Standardization of the audit events and values they include would be beneficial. The plugin would be able to extract audit data more efficiently without using complicated data structures and any ambiguities about what information that the audit events carry would disappear.

Furthermore, there is another plugin that stores some of the audit events in a database. It is able to provide time-related statistics for the reactive plugin by executing complicated SQL queries. Based on the retrieved information from the database, the reactive plugin can make certain decisions. This extends the functionality of the whole mechanism since it is possible to detect anomalies.

The Chapter 6 comes up with a couple of examples where the two plugins take part. Firstly, the reactive audit can be used to reflect the current state of a system by adding or deleting audit rules. This is useful especially when there is a need to control each user of a system by different set of audit rules. The number of audit rules in the kernel influences the system performance as is discussed in the previous chapter. Secondly, the plugins are able to reveal suspicious activities as well as anomalies and send warnings to an administrator. Finally, the reactive plugin could be configured to ban users or kill processes that represent a security risk for the system. The implemented mechanism meets all the functional requirements that were described in the proposal.

The future work on the reactive plugin involves optimization of reaction evaluation. Furthermore, the commands within a reaction definition creates new processes that are executed asynchronously. It might be better off creating a mechanism for synchronous execution of these commands. There is also some work on the plugin for creating statistics. If an anomaly is detected, the plugin should stop storing audit events that caused the anomaly. If these events are stored constantly, they may cause that the same anomaly will not be detected in the future.

To sum it up, the thesis achieved its goal of implementing the mechanism of the reactive audit. The mechanism extends the functionality of the audit system in Linux, it is general enough to be used in many different ways.

Bibliography

[1] Bishop, M.; Dilge, M.: Checking for Race Conditions in File Access, volume 9, pages 131–152. Department of Computer Sience, University of California at Davis, Davis, CA 95616-8562, 1996.

[2] Brenton, C.: Auditing Windows NT. Available at URL: `http://www.arcert.gov.ar/webs/textos/ntaudit.pdf` (May 2010).

[3] Grubb, S.: Native host intrusion protection with RHEL5 and the audit subsystem [online]. San Diego, May 2007. Red Hat. Available at URL: `http://people.redhat.com/sgrubb/audit/summit07_audit_ids.pdf` (May 2010).

[4] Horman, N.: Understanding and programming with netlink sockets [online], December 2006. Available at URL: `http://lovezutto.googlepages.com/netlink.pdf` (May 2010).

[5] Levine, J.: Flex & Bison. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st edition, 2009. ISBN 978-0-596-15597-1.

[6] Linux manual page. Audisp-prelude - plugin for idmef alerts [online]. Available at URL: `http://linux.die.net/man/8/audisp-prelude` (May 2010).

[7] Linux manual page. Audisp-remote - plugin for remote logging [online]. Available at URL: `http://linux.die.net/man/8/audisp-remote` (May 2010).

[8] Linux manual page. Audispd - an event multiplexor [online]. Available at URL: `http://linux.die.net/man/8/audispd` (May 2010).

[9] Linux manual page. Audispd-zos-remote - z/OS Remote-services Audit dispatcher plugin [online]. Available at URL: `http://linux.die.net/man/8/audispd-zos-remote` (May 2010).

[10] Linux manual page. Auditctl - a utility to assist controlling the kernel's audit system [online]. Available at URL: `http://linux.die.net/man/8/auditctl` (May 2010).

[11] Linux manual page. Auditd - the Linux audit daemon [online]. Available at URL: `http://linux.die.net/man/8/auditd` (May 2010).

[12] Linux manual page. Aureport - a tool that produces summary reports of audit daemon logs [online]. Available at URL: `http://linux.die.net/man/8/aureport` (May 2010).

[13] Linux manual page. Ausearch - a tool to query audit daemon logs [online]. Available at URL: `http://linux.die.net/man/8/ausearch` (May 2010).

[14] Linux manual page. Autrace - a program similar to strace [online]. Available at URL: `http://linux.die.net/man/8/autrace` (May 2010).

[15] Linux manual page. System - execute a shell command [online]. Available at URL: `http://linux.die.net/man/3/system` (May 2010).

[16] Mauerer, W.: Professional Linux Kernel Architecture, pages 1097–1116. Wiley Publishing, Inc., 10475 Crosspoint Boulevard, Indianapolis, IN 46256, 2008. ISBN 978-0-470-34343-2.

[17] Mitchell, M.; Oldham, J.: Advanced Linux Programming. New Riders Publishing, 201 West 103rd Street, Indianapolis, IN 46290, 1st edition, 2001. ISBN 0-7357-1043-0.

[18] Project page. The page of SQLite project - FAQs. Available at URL: `http://www.sqlite.org/faq.html` (May 2010).

[19] Project page. The page of the Linux audit project [online]. Available at URL: `http://people.redhat.com/sgrubb/audit` (May 2010).

[20] Project page. Syslog: Main/Home page [online]. Available at URL: `http://www.syslog.org/` (May 2010).

[21] Security standard. Common Criteria for Information Technology System Evaluation, part 2: Security functional components [online]. Available at URL: `http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R3.pdf` (May 2010).

[22] Watson, R.; Salamon, W.: The FreeBSD Audit System. Available at URL: `http://www.trustedbsd.org/20060303-ukuug2006lisa-audit.pdf` (May 2010).

[23] Wells, S.: The Linux Audit Subsystem: Deep Dive [online]. Denver, August 2009. Red Hat. Available at URL: `http://linuxvm.org/present/SHARE113/S9203sw.pdf` (May 2010).

Appendix A

Content of CD

The enclosed CD contains the following:

- source files for generation of this document
- GIT repository of the audit
- patches for the audit