Rule-Based Distributed Intrusion Detection

Abdelaziz Mounji

Institut d'Informatique
University of Namur
rue Grandgagnage 21,
B-5000 Namur Belgium
Email: {amo}@info.fundp.ac.be

July 11, 1997

Contents

[1]To be reworked.

List of Figures

List of Tables

Chapter 1

Introduction

Computers and network computers of today's organizations and academia are increasingly becoming the targets of computer crimes which can result in loss of productivity, competitive advantage, theft of corporate business information, and stealing of research data. Considering the reliance of humans on computers and network infrastructures to perform virtually every aspect of day to day activity, there is a critical need for ensuring the reliability and integrity of these infrastructures. The proliferation of large-scale network connectivity has further complicated the task of protecting our infrastructures against security infractions and computer criminals. Once considered as teenage fun games, today's computer security break-ins are performed by determined criminal organizations, which have sufficient resources and expertise.

Computer and network security have seen a rising research interest within the past two decades in order to increase the capability of our security infrastructures to address the new challenges and security risks facing corporate and research communities. The security infrastructure provides several security services such as *privacy, integrity, availability*, and *authentication*. The privacy security service ensures the non-disclosure of data stored in a computer or transmitted over a communication medium. The integrity security service allows one to protect data against unauthorized modification and destruction. The availability security service protects against denial of service attacks by preventing malicious users from exhausting the resources available to legitimate users of the system. These malicious users intentionally consume an abnormally large portion of available resources, thereby denying access to other users. The authentication security service ensures that a person or a program identified by the system is actually what it claims to be. This is an important service which allows the system to grant access only to authorized users and programs for the requested data or service. Cryptography is an indispensable part of computer and network security on which the privacy, integrity, availability and authentication services are based.

The security services hence provide preventive measures for ensuring the security of the system because they help avoid security violations to occur in the first place. By contrast, intrusion detection is an after the fact approach to system and network security, which aims at detecting security intrusions by monitoring the actual use of the system by users and application programs. The purpose of the thesis is to present a novel intrusion detection approach based on intelligent analysis of security audit trails. The system uses a powerful rule-based language allowing the site security officer to efficiently analyze large audit files which record user and system activity. The thesis also presents an implemented distributed system for detecting intrusion scenarios in a multi-host heterogeneous environment. The presented system is universal in the sense that it allows us to analyze the audit trails of different operating systems and uses format adaptors to translate native audit trails into a canonical format.

1.1 Thesis Statement

Since it is difficult to achieve an exhaustive model of intrusions and because of the inadequacy of existing auditing mechanisms, the approach proposed in this thesis pursues pragmatic design goals, in order to better address the audit trail analysis problem. The approach relaxes assumptions about the audit data, both at the syntactic and semantics levels. It is based on a powerful and efficient computational framework using a rule-based language especially crafted for the problem of matching arbitrarily complex event sequences. In this framework, only the inherently sequential nature of audit trails in particular and event stream in general is retained. Although the chronological nature of events is useful for optimizing detection rules, it is not required explicitly by the rule-based language. Relaxing assumptions about the analyzed audit data allows the tool to be applicable to all existing and future audit trails. The rule-based language viz. RUSSEL (RUle BaSed Sequence Evaluation Language) is an amalgamation of procedural and rule-based programming paradigms. The main features of the language can be outlined according to three major requirements: universality, power, and efficiency.

Universality of the analysis tool is attained by designing a normalized audit file format (NADF) which is sufficiently flexible to ensure that all existing audit trails can be translated to it in a straightforward way. This format allows audit records to be self-identifying, so that they are easily interpreted in the language. The NADF format does not necessarily involve rigid semantic interpretation of audit records in terms of subject, object, event type, timestamp, etc. Audit records are considered as uninterpreted strings of bytes the meaning of which is considered only at the language level. This is in contrast with other approaches where the audit records are assumed to include specific fields that are related to the model of intrusions. However, in case there is not a one-to-one mapping between native audit records and canonical audit records[1], the preprocessing may be complicated. For this reason, and in order to accommodate the disparity in audit record formats, the approach proposed in this thesis handles the semantics differences in audit data at the language level in stead of at the preprocessing phase.

Power of the analysis tool is attained thanks to the rule-based language which allows one to express complex queries characterizing the sequential relationship of audit record patterns. Since attack scenarios may be complex and since their manifestation in the audit trail are dependent on the operating system, the power of the language makes it possible to accommodate the complexity and variations in attack signatures. This is in contrast with the approach of providing ad-hoc queries of a data base management system which may not be powerful enough to express complex patterns of system abuses.

Efficiency of the analysis tool is achieved based on predefined control structures which are built into the language and which are suitable for the specific nature of event stream analysis. The efficiency is also addressed by means of a careful implementation of the language. The efficiency feature of the language allows one to perform a single pass analysis of the audit trail and makes real-time analysis feasible as will be demonstrated by experimental results.

The purpose of this thesis is to describe the intrusion detection system ASAX (*Advanced Security audit trail Analysis on uniX*), which is based on the above three features.

[1] For instance, when the subject field must be reconstructed from several records related to the same job.

1.2 Structure of the Thesis[2]

The thesis is structured as follows. Chapter 2 contains an overview of related works in intrusion detection. Chapter 3 describes the rationales behind the design of the intrusion detection system ASAX presented in this thesis. In Chapter 4, we informally introduce the salient features of the rule-based language RUSSEL underlying ASAX through examples. Chapter 5 provides a precise and complete description of RUSSEL at a machine independent level. In Chapter 6, we provide a description of the machine and operating system dependent aspects of the language for the UNIX operating system. Chapter 7 shows how finite state automata can be simulated using the language RUSSEL. In Chapter 8, we present example RUSSEL programs for detecting intrusion patterns under Solaris 2 Basic Security Module (BSM). Chapter 9 describes a continuous configuration analysis system, which is coupled with ASAX, thereby enabling a dynamically adaptive detection rule set. Finally, Chapter 10 aims at addressing the problem of detecting intrusion patterns in a multi-host environment.

[2] To be reworked.

Chapter 2

Related Work in Intrusion Detection

2.1 Introduction

The previous chapter introduced classical approaches to computer security. Theses approaches provide mechanisms for data confidentiality, data and communication integrity, and assurance against denial of service. Unfortunately, building provably secure computer systems is extremely difficult or even impossible for the following reasons:

- The complexity of current operating systems, network services, and application programs makes it extremely difficult to formalize them exhaustively in an attempt to prove their compliance with the security requirements. In particular, it is notoriously difficult to ensure that a cryptographic protocol actually provides the goals it is designed to achieve [1]. Experience has shown that even published cryptographic protocols, which are recommended by experts, have suffered protocol failures [2, 3, 4]. Moreover, cryptographic protocols may be made inoperable in case the cryptographic key is stolen or lost, and cannot defend against weak passwords.

- Completely secure systems are likely to impose considerable constraints for users whose primary goal is to obtain fast access and flexible sharing of data, especially in an academic environment [5], let alone the potential overhead in performance. Indeed, users often complain about being required to carry along a list of one-time passwords for S/Key access, and having to authenticate to several intermediate proxy servers[1] before eventually using a service.

- Completely secure systems are still vulnerable to authorized users who misuse their privileges (the *insider threat*). For instance, a security administrator can turn off auditing for selected users (including himself), create back-doors thereby circumventing the protection mechanism, or access user private data.

Considering the fact that the prevention-based approaches cannot be applicable or completely relied upon, it becomes reasonable that additional complementary approaches have to be devised in order to enhance the security of a computer system. In the mid-80s, the intrusion detection approach emerged as a supplement to conventional security mechanisms, which attempt to detect misuses and anomalies by observing users and system's actual activity [6, 7, 8, 9]. The benefit of the intrusion detection approach is to provide the security officer with the ability to check if a

[1] Especially when the server is located behind a firewall.

would-be attacker has bypassed the security shield provided by the security protection mechanisms. Moreover, given the inevitable security-flaws in the protection mechanism, an intrusion detection system can be a valuable tool in identifying and eliminating these flaws. The intrusion detection approach is founded on the assumption that it is possible to actually distinguish normal activity from an abnormal or a misuse behavior. Consequently, addressing the problem of detecting misuse and anomalies of a computer systems requires 1) a good understanding of what constitutes a misuse or anomalous behavior, 2) a reliable recording of user activity, and 3) a technique for analyzing user activity records.

The rest of this chapter is organized after the above requirements. In Section 2.2, we categorize computer security intrusions based on the paper by Anderson [10]. In Section 2.3, we define the notion of auditing mechanism. In Section 2.4 we describe the two main approaches to intrusion detection: *anomaly detection* and *misuse detection*. Section 2.5 contains an overview of related works in anomaly detection and Section 2.6 overviews related works in misuse detection. In Section 2.7, we outline the major problems with current intrusion detection. Finally, Section 2.8 is a brief history on the development of the intrusion detection tool proposed in this thesis.

2.2 Computer Security Threats

The earliest work on intrusion detection was carried out by Anderson [10] who identified three major categories of threats to a computer system:

- external penetrators (who are unauthorized users of the system);
- internal penetrators (these are legitimate users of the system who access computer resources for which they are not authorized). Internal penetrators can be further divided into:
  - masqueraders (who steal the identity of other users thereby acquiring the privileges of those users);
  - clandestine users (who have succeeded in evading the auditing mechanism, either by operating at a sufficiently low level, or by disabling the auditing mechanism altogether);
- misfeasors: these are authorized users who misuse their privileges.

Anderson suggested that audit trail analysis can address some of these threats: external penetrators can be detected by monitoring the audit trail for abnormally high rate of failed login attempts; masqueraders can be detected by significant deviations in normal user behavior profiles. However, he argued that clandestine users can be extremely difficult to detect since they circumvent or corrupt the auditing mechanism itself. A very low level auditing can nonetheless reduce the risk of clandestine users evading the auditing. Moreover, writing audit data to a write-once device can also be considered as an attempt to preventing audit data corruption. In NIDES [11], audit data is sent to a dedicated host where the analysis takes place thereby reducing the chance of audit data corruption. In NIDES two approaches were suggested for detecting users abusing their privileges (misfeasors). On the one hand, NIDES' expert-system uses *a priori* rules for detecting "socially unacceptable" behaviors that are analogous to those that characterize intrusions. On the other hand, NIDES defines user groups on the basis of user roles or job assignments with established normal behavior. The norm is compared with user's behavior to detect abuse of privilege [12].

2.3 Auditing Mechanism

Security auditing is the process of collecting user activity records in what is called an *audit trail*. Audit trails consist of a chronological sequence of audit records, each of which describes the occurrence of a security-related event. An event reports the attributes of an action performed by a subject on an object. These attributes include the type of the event, the identity of both the subject and the object involved, the resource consumed by the action, and whether the action was successful or not. Consequently, audit trail analysis potentially allows a security officer to reconstruct the history of what happened in the system. In particular, audit trail analysis can address the problem of detecting penetration of the system and the misuse of system resources. Auditing mechanism is a feature provided by trusted operating systems. The Department of Defense of the United States defined a set of criteria — known as the Orange Book — that allows one to evaluate the security level of a computer system [13]. In this classification, there are four divisions from D to A, in an increasing order of security levels. The evaluation criteria are structured into four major sets: security policy, accountability, assurance, and documentation. Computer systems belonging to division C are required to provide a discretionary protection; those in division B must provide a mandatory protection; those in division A must provide formal means for *verifying* protection enforcement. Finally, division D is the lowest protection level and is reserved for those systems that have been evaluated but that fail to meet the requirements for a higher evaluation class. Division C and B are further divided into the classes C1, C2, and B1, B2, and B3. All computers ranked at the level C2 or higher must provide security auditing mechanisms. Although the Orange Book has defined these criteria as a basis for evaluating the security level of computer systems, it has provided no specific requirements as to how to implement the protection mechanism. The implementation design rests with each vendor as long as it complies with the recommendations. Particularly, these criteria do not specify the exact format of audit records, and more importantly, it does not define how audit trails should be used for the purpose of intrusion detection.

2.4 Approaches to Intrusion Detection

There are two main approaches to intrusion detection: anomaly detection and misuse detection. The anomaly detection approach defines a model of normal activity by profiling the user activity; any significant deviation from the established norm is considered anomalous. By contrast, the misuse detection approach defines what specific user's actions constitute a misuse and uses *a priori* rules for encoding and detecting known intrusion patterns.

Since masqueraders generally produce activity profiles which deviate from legitimate users' activity profiles, the anomaly detection approach is well suited for detecting masqueraders. However, if a normality threshold is set too low, a normal activity may be flagged as intrusive; this results in a so-called false positive alarm. Similarly, if the normality threshold is set too high, an intrusive activity may be flagged as normal; this results in a so-called false negative alarm.

Alternatively, the misuse detection approach is better suited for detecting external penetrators and misfeasors, through the use of *a priori* rules for detecting each intrusion pattern or site security policy violation. For instance, external penetrators produce repeated login attempts and misfeasors exploit known security vulnerabilities, violate the local site security policy, or follow known intrusion patterns. However, since the set of *a priori* rules may not be comprehensive enough to ensure the coverage of all existing and future intrusion patterns, certain intrusions may not be reported by a misuse detector, resulting in a false negative alarm.

Denning [8] proposed a model of intrusion detection that can be viewed as a generalization of most existing intrusion detection systems built to date. The model is based on the hypothesis that security violations can be detected by monitoring a system's audit trail for abnormal patterns of system usage. The model is independent of any particular system, application environment, system vulnerabilities, or type of intrusions. Central to the model is the abstract concept of activity profile, which characterizes the behavior of subjects with respect to objects. An activity profile is a description of normal activity in terms of statistical metrics: the event counters metric accounts for the number of audit records occurring during a period of time and satisfying some property; the internal timer metric measures the time separating related audit records; the resource usage metric quantifies resources consumed by events. Each of these metrics can be used with a statistical model that characterizes the normal distribution of the metric. Proposed statistical models for the above metrics are: operational model, mean and standard deviation model, multivariate model, Markov process model and time series model. The reader is referred to [8] for a detailed definition of these models. Activity profiles are updated progressively to track the change in user and system behavior. Upon receiving an audit record, the system updates the variable distribution contained in relevant activity profiles and also checks for abnormality. The rule-based component of the model is composed of activity rules, which specify an action to be taken when an audit record is anomalous. Such actions include generating anomaly records, summary reports. etc.

SRI International developed an intrusion detection system, viz. NIDES [11], implementing this model, which uses an extensive number of statistical profiles both for users and system.

2.5 Anomaly Detection Approach (Profiles)

This approach consists of establishing normal behavior profile for user activity and observing significant deviations of actual user activity with respect to the established normal pattern. Significant deviations are flagged as anomalous and should raise suspicion. Ideally, the daily use of a computer system by a given user follows a recognizable behavior pattern, which can serve as a characterization of the user identity. The set of application programs, commands, and utilities invoked by a user is often determined by the nature of the job assignment for that user. For instance, a secretary is expected to use a document processing program, e-mail, or calendar management applications. By contrast, a programmer almost always restrict her/himself to editing source files, compiling and testing programs and so forth. Clearly, the fact that a compiler is suddenly invoked by a secretary should be considered as highly suspicious and hence should be immediately brought to the attention of the security officer who may conclude that an intruder is probably masquerading as the secretary. Moreover, even though a group of users may have the same job assignments, it is still possible to distinguish between their profiles based on their personal habits, such as their session location, normal working hours, or the pattern of commands they frequently use.

In addition, since a user's normal activity profile may change over time as the user learns new commands and achieves higher qualifications, his/her established profile must also be adapted continuously to accommodate the change in the behavior. A statistical profile is a set of metrics M_1, ..., M_n each of which being related to a particular aspect of his or her behavior. Example metrics include CPU usage, I/O usage, frequently accessed files, error rate, etc. Each metric contained in the pattern is also associated with a threshold or a range beyond which the activity is considered anomalous. Determining whether a given audit record indicates a deviation with respect to recent pattern of activity amounts to verify that the reported activity exceeds the threshold or falls outside the acceptable range.

The benefits of the anomaly detection approach are:

- Ability to detect unknown intrusion scenarios since it does not make a priori assumption about the specific steps taken by an intruder in exploiting flaws in the protection system. Although the intruder may use different penetration strategies in defeating the protection mechanisms, the statistical anomaly detector will still be able to flag his or her activity as suspicious only based on departure from established normal pattern. Obviously, closer examination or review of the audit trail may further investigate the specific attack scenario used by the attacker;

- Assuming that the set of metrics are carefully chosen, are comprehensive enough to provide a good discrimination of the normal activity, and are adequately adapted to the evolving pattern of behavior, the statistical anomaly detection approach does not require repeated modifications in order to remain effective.

The main drawbacks of this approach can be summarized as follows:

- The exact order of event occurrence is not taken into account; only the final cumulative measure is considered. Consequently, an anomaly which is dependent on the sequential relationship of events may be missed.

- The process of dynamically adapting the statistical profile as the user behavior changes over time may be abused by an attacker if s/he is aware of being monitored and succeeds in gradually training the profile thereby avoiding detection by a purely statistical anomaly detector. However, the time series as proposed in [8], will be effective in detecting significant shift in activity profile over a period of time. The time series takes into account the order and inter-arrival times of the observations. An observation is abnormal if the probability of occurring at that time is too low. Computing time series is however computationally expensive.

- Choosing an appropriate threshold or range of normal values for metrics is a difficult task. Lower thresholds may result in an increase in false alarm rates while higher thresholds may miss anomalous activity. This is a particular issue if a real-time response capability is supported in the intrusion detector. In case a legitimate process is recognized as being suspicious, the anomaly detector may erroneously kill such a process or terminate the corresponding user session completely. Responses to false alarms can be even more dangerous if the response has a more radical effects such shutting down network services or halting the operating system itself.

- Finally the completeness of the statistical anomaly model has still to be verified, i.e., given a set of selected metrics, it is unclear whether all anomalous behavior can be captured using these metrics and corresponding thresholds. This is obviously a challenging problem from a theoretical viewpoint.

In the following sections, we describe previous works on intrusion detection following mainly the anomaly detection approach; some of them however, combine the misuse detection and the anomaly detection approaches.

2.5.1 Neural Networks

A neural network consists of many simple, neuron-like processing elements called *units* that interact using weighted connections [14]. The knowledge in a neural network is encoded in the structure of the neural net, in terms of the connections between units and their weights, so learning consists of changing the weights or adding or removing connections. Neural nets have been advocated as a viable approach to intrusion detection in [15, 16, 17, 18]. In a first stage, the neural net is trained by sample patterns that are representative of normal user behavior. During a second stage, the neural net receives user activity data and determines to what extent this activity exhibits similarities with the samples it was trained on. Abnormal data yields a significant change in the state of units, their connections, or weights. Neural nets are capable of adaptively learning the model of normal user and system's activity based only on the activity data itself. In particular, they do not make prior assumptions on the expected statistical distribution of measures, nor do they employ a fixed set of features describing aspects of user behavior. Hence, a neural network-based anomaly detector overcomes the difficulty of devising statistical features, and consequently, the problem of selecting a *good* subset of features is irrelevant.

The major drawback of a neural network approach in intrusion detection and in learning in general is that they do not provide explanatory information for detected anomalies. Assuming that an audit record is recognized as being anomalous, the neural network will modify its notion of normality in stepwise corrections, rather than by providing explanation for the reason of the anomaly. Therefore, neural nets cannot be considered as a replacement for the statistical profile-based anomaly detection, but as a valuable addition. Nonetheless, neural nets can provide supporting evidence for the effectiveness of a statistical model by means of simulation. A prototype anomaly detection system using a neural network approach, which is coupled with an expert system for further investigations of anomalies reported by the neural network is described in [15] . The application of neural networks was also investigated in [16, 17, 18]. We do not know, however, how successful these approaches have been in dealing with high volume of audit data.

2.5.2 Predictive Pattern Generation

Whereas intrusion detection expert systems utilize a fixed set of detection rules, a predictive pattern generation-based system [19, 20] uses a dynamic set of rules for detecting intrusions. These rules are inductively generated based on the sequential relationships and temporal properties of the observed events. By identifying regularities in past observed event sequences, the inductive engine is able to infer that some specified event types are more likely to occur next in the input stream than other types of events. The inductive engine assigns a probability to each most likely event. Inductively generated rules can be written in the form:

$$\mathrm{E}_1, ..., \mathrm{E}_k \ :- \ (\mathrm{E}_{k+1}, \mathrm{P}(\mathrm{E}_{k+1})), ..., (\mathrm{E}_n, \mathrm{P}(\mathrm{E}_n)).$$

which reads: assuming that the input stream contains the event sequence E_1, ..., E_k, the events E_{k+1}, ..., E_n are the more likely to be seen in the rest of the input stream with probabilities $\mathrm{P}(\mathrm{E}_{k+1})$, ..., $\mathrm{P}(\mathrm{E}_n)$.

According to this method, if a sequence of events matches the head of a rule, then the next event is considered anomalous if it is not one of the predicted events in the body of the rule. Moreover, the learning capability of the system is attained by deleting less predictive rules from the rule base. A rule is more predictive than another rule if it successfully predict more events than the other. Detection rules may also be static as is the case in conventional expert systems. In a sense, the dynamically generated rules encode user normal behavior profiles. The effectiveness of the approach depends on training the system on patterns that are the most representative of the normal user behavior. Consequently, a primary weakness of this approach is that the inductively generated rules may not be comprehensive enough to cover all possible normal user behavior. In other words, some event patterns may be considered anomalous simply because they do not match the head of any rule.

The strengths claimed for this approach are [21]:

- better handling of users with wide variances of behavior but strong sequential patterns;
- ability to focus on a few relevant security events rather than the entire login session that has been labeled suspicious;
- better sensitivity to detection of violations. Cheaters who attempt to train the system during the learning phase can be discerned more easily because of the semantics built into rules.

2.5.3 Data Clustering

The problem of anomaly detection can be viewed as the general problem of data classification, in which the analyzed data is organized into *clusters* according to some clustering criteria. In statistical anomaly detection, a user's activity data is organized into two distinct clusters: anomalous activity cluster and normal activity cluster. Statistical clustering uses probability distribution of activity records. Neural networks clustering adaptively learn normal behavior through sample record patterns. Other clustering criteria include distance clustering using a distance measure to establish membership in a given cluster. Conceptual clustering requires that the objects satisfy necessary and sufficient conditions for cluster membership [22]. In [23] the reader can find a comprehensive description of several other clustering methods and algorithms.

2.5.4 HAYSTACK

HAYSTACK [24] was developed by Los Alamos National Laboratories (LANL) for the US Air Force on Unisys 1100/60 mainframes and is used for reducing voluminous audit trails to summary reports of intrusions and security incidents [24]. It also provides detection of masqueraders, penetrators, leakage of information, denial of service, and attempted break-ins.

HAYSTACK maintains a database of user and user group profiles in an Oracle database with embedded SQL and runs on a physically secured PC. Audit data is collected and preprocessed to a Canonical Audit File format (CAT files) on the mainframe and is then transfered from the mainframe to the PC by manual tape or by electronic file transfer.

HAYSTACK uses two methods in detecting misuse and anomaly:

Pattern-based analysis uses a record-wise selection of important events involving a modification of the access control mechanism, user-ids, group-ids, etc. The system also allows the security officer to increase auditing of selected users, groups of users, or files.

Statistical-based analysis the statistical anomaly detection in HAYSTACK employs two different methods. The first method evaluates to what extent a user session resembles one of predetermined intrusion types. In this method, the system maintains two dozen measures of user behavior such as time of work, number of created files, number of printed pages, etc. Given a particular type of intrusion, the system associates a weight with each statistical measure, which describes the usefulness, or relevance, of the measure in detecting the given intrusion type. A weighted intrusion score is calculated by summing the weights associated with all measures for which the threshold was exceeded during a user session. This weighted intrusion score is then used to assign a *suspicion quotient* to the session, based on the knowledge of the distribution of the weighted intrusion scores for all previously observed sessions. A detailed presentation of the algorithms used for computing the suspicion quotient is in [25].

The second statistical anomaly detection method employs the common technique of detecting deviations in a user's session profile from the normal established profile for the user's session.

The effectiveness of the suspicion quotient in detecting intrusions depends on adequate choice of the weights associated with each statistical measure and on the number of intrusion types which are covered by the system. Moreover, the pattern-matching based analysis does not take into account the sequential relationship between events since the selection is record-wise. The manual tape transfer of audit data makes real-time analysis impossible and hence, of course, the system lacks the ability to abort abusive user activity.

2.5.5 MIDAS

The MIDAS (Multics Intrusion Detection and Alerting System) is an expert system which provides real-time intrusion and misuse detection for the National Computer Security Center's networked mainframe, Dockmaster, a Honeywell DPS-8/70 Multics [26]. The basic design of MIDAS is influenced by the generic intrusion detection model proposed by Denning [8]. MIDAS was developed using the Production-Based Expert System Toolset (P-BEST), an in-house expert system shell that provides powerful rule sets using a bottom-up inference engine. MIDAS employs statistical profiles for user sessions as well as for system-wide behavior and uses three types of heuristics, or rules, to detect user's and system's misuse:

Immediate attack rules: these rules perform record-wise selection of audit records that exhibit suspicious behavior regardless of the rest of the audit trail.

User anomaly: these rules make use of the statistical profiles to detect anomalous user behavior in user sessions. These profiles are updated upon session termination.

System state: these rules are similar to the user anomaly detection rules, however, the employed statistical measures are related to system-wide parameters rather than to a single user.

2.6 Misuse Detection Approach

The misuse detection approach consists of searching the audit trail for the occurrence of specific patterns of audit records generated by known misuses of the system with respect to past intrusions, known vulnerabilities, and security policies [7, 27]. While in anomaly detection we model the normal behavior of users and we observe possible departures from this normal behavior patterns, in misuse detection, we encode a priori knowledge of what specific actions as manifested in the audit trail constitutes a misuse of the system. The pattern of audit records representing the penetration scenario is also referred to as the penetration or intrusion *signature*.

2.6.1 Rule-based Expert Systems

The misuse detection problem is often addressed by the use of a traditional, general purpose rule-based expert system. In such systems, there is a separation of the declarative knowledge related to problem domain from the inference engine which performs a reasoning about the fact base.

The basic idea is that every detection rule in the rule base encodes a particular scenario and is responsible for detecting its occurrence by pattern matching the corresponding intrusion signature against the audit trail. However, intrusion detection rules can be made general enough to capture slight variations of the same penetration obtained by using a reordering of audit record sequences, or by aliasing certain commands. The intrusion detection system IDES [27, 28] uses a general rule-based expert system P-BEST, while the intrusion detection system OSIRIS [29] uses Prolog rules.

2.6.2 State Transition Analysis

State Transition Analysis is an approach to misuse detection using state transition diagrams for representing and detecting known penetration scenario [30]. The system uses a scheme similar to finite state machine graphs [31]. In this approach, a penetration is modeled as a sequence of actions performed by an attacker that leads from some initial state on a system to a target compromised state. The initial state corresponds to the state of the system just prior to the execution of the penetration, and the compromised state corresponds to the state of the system resulting from the completion of the penetration. USTAT [32] is a real-time intrusion detection system that implements this approach for a Unix system.

USTAT uses a canonical record structure, which is similar to that of Denning's intrusion detection model [8]. USTAT's canonical format is defined by the triple: ⟨*subject*, *action*, *object*⟩ meaning *subject* performs the *action* on the *object*. The subject attribute is composed of the real user Id, the effective user Id, and the group Id of the user who performs the action. The action attribute is composed of the action type, the process Id, and a time stamp. The object attribute is composed of several further attributes which are the object name, permission modes, owner, group owner, inode number, device number, file system Id, and host name. Target system-specific audit records produced by a particular auditing facility (such as the Basic Security Module of the SunOS operating system) are transformed into the above canonical format before being supplied to the inference engine component of USTAT. Action types that are considered in USTAT are those actions that manipulate system objects and/or modify user privileges. The action field contains one or more of the values: Read, Write, Create, Delete, Execute, Exit, Modify_Owner, Modify_Perm, Rename, and Hardlink.

A state in a state transition diagram is a list of state assertions describing the system state in terms of system attributes and/or user privileges. A state assertion is a conditional expression composed of primitive assertions and using logical AND/OR/NOT connectives. The primitive assertions are built into USTAT and include the following example primitives:

owner(*file_var*) = *user_id*
: Evaluates to true if the owner of *file_var* is the *user_id*.

euid = *user_id*
: Evaluates to true if the effective user id of the subject of the audit record being processed equals the *user_id*.

permitted(*perm*, *file_var*)
: Evaluates to true if the permission bit given as *perm* is set in *file_var*'s permission bits.

exist(*file_var*)
: Evaluates to true if *file_var* exists in the file system.

shell_script(*file_var)*
Evaluates to true if *file_vars* is a shell script, i.e., it starts with the string `"#!/bin/sh"`.

A transition in a state transition diagram is labeled by a user action, also called *signature action*. Penetrations are detected in USTAT using state transitions diagrams as follows. At a given time, a number of penetration scenarios have being partially matched in the audit trail, which means that some sequence of signature actions have caused the system to reach some state in each individual state transition diagram. If the final state of any one transition diagram has been reached, the inference engine concludes that the penetration scenario corresponding to that state transition diagram has been successfully completed. Otherwise, for each state transition diagram, the inference engine expects the signature action leading out from the current state and for which the state assertions of the next state hold. If such action occurs for any state transition diagram, the inference engine places the system in the next state. Obviously, if the occurrence of a signature action is such that the state assertions of the current state no longer hold, the inference engine moves the state transition diagram back to the nearest state, the assertions of which still hold. If a penetration is detected, the relevant information gathered along the transitions are passed to the decision engine, which provides the security officer with a report describing the perpetrated penetration.

In order to cover all variants of a particular penetration scenario that are obtained using different links to the same physical file, USTAT also maintains for each security critical file a list, called *hardlinks information*, of all links made to it. The hard link information is updated accordingly by observing relevant signature actions that manipulate sensitive files. The result is that all references to a physical file through any of its links are treated in the same way. Moreover the system supports permutable transitions[2] by specifying a transition dependency field for each transition. The latter field indicates for each transition (a possibly empty) list of transitions that must have been fired before the transition can fire.

USTAT is able to track multiple instantiations of a particular state transition diagram in order to detect all instances of the corresponding penetration scenario that are perpetrated by several users.

In the following, we present an example penetration scenario and show how it is represented and detected using the STAT approach.

Example penetration scenario

One of the possible ultimate goals of an attacker is to obtain a copy of the interactive shell interpreter `/bin/sh`, which is owned by root, setuid enabled, and executable by everyone. This is because all commands supplied to such a copy of the interactive shell are executed under the identity of root, thereby giving the attacker root privileges. Note that if a shell copy is owned by root, is executable by everyone, but is not setuid enabled, the interactive shell runs under the identity of the user who invoked the shell, and not that of root. On the other hand, any user has the ability to create a setuid enabled copy of the shell, which is executable by everyone. However, in the latter case, the privileges of the user are unchanged since the created copy is owned by that user and consequently, the interactive shell runs under the identity of that user. The attacker has then to find a means of changing the ownership of the created shell copy to that of root. In the following, we explain a

[2]Two state transitions are said to be permutable if the successful completion of the penetration scenario does not depend on the order of the transitions in the state transition diagram.

particular way of achieving this goal by exploiting a security vulnerability that existed in the UNIX 4.2BSD mail program.

When a user sends a mail message, the mail delivery program appends the mail message to the recipient's mail box (which is the file `/usr/spool/mail/username`) and then changes the ownership of the recipient mail box to that of the recipient. In particular, when a user sends a mail message to root, the mail delivery program appends the mail message to the file `/usr/spool/mail/root` and then changes its ownership to that of root. Hence, the ultimate goal of the attacker now reduces to creating a setuid copy of the shell, which is executable by everyone, and which is named after root's mail box. Consequently, since the attacker is not authorized to override root's mail box, s/he may simply wait until root has no mail messages (in which case, the file `/usr/mail/spool/root` does not exist) and then creates a shell copy under the name `/usr/spool/mail/root` [3]. The vulnerability existed in UNIX 4.2BSD because the mail delivery program fails to disable the setuid bit before changing the ownership of the mail box.

Now assuming that root has no mail waiting, one way to obtain a setuid copy of the shell, which is owned by root and executable by everyone is to type the following command sequence.

```
> cp /bin/sh /usr/spool/mail/root
> chmod 4755 /usr/spool/mail/root
> touch x
> mail root < x
```

The first command creates a shell copy under root's mail box and the second command sets the setuid bit of the created copy. The third step is not vital to the successful completion of the penetration; the attacker just needs a bogus file to send to root as a mail file. Note that this step may occur before step 1 or before step 2 provided that it occurs before step 4. In the latter step, the attacker sends the bogus mail file `x` to root, and the result is that the mail program changes the ownership of root's mail box to root without reseting its setuid bit. The bogus message at the end of the shell will be treated as the symbol table of the file and will not affect the proper execution of the shell.

It is worth noting that in this particular scenario, the attacker has exploited the mail vulnerability as a means of changing the ownership of a setuid file to that of root, but any other means of achieving the same goal would have been acceptable as well.

The state transition diagram used by USTAT for representing and detecting the above described penetration is shown in Figure 2.1, in which the circles denote states with a list of state assertions below them, and where transitions are represented by directed arcs connecting states and labeled by signature actions. In this diagram, the signature action labeling the last transition has the effect of changing the ownership of the file `/usr/spool/mail/root` and the state assertions of the final state mean that the latter file is setuid enabled and is owned by root. The first state requires that the user performing the signature actions is not root, otherwise there is no need to gain root privileges. It is important to note that if the setuid bit is reset by the last signature action while the system was in state S3, the inference engine will transition the system back to state S2. This transition is not represented explicitly in the diagram, but is rather accounted for internally in the inference engine. In general, if the last signature action makes the state assertions of a given state no longer hold, an implicit transition labeled by that signature action and leading from that

[3] This is possible since the mail spool directory `/usr/spool/mail` is writable by everyone and has the sticky bit set, meaning that nobody can delete somebody else's mail file.

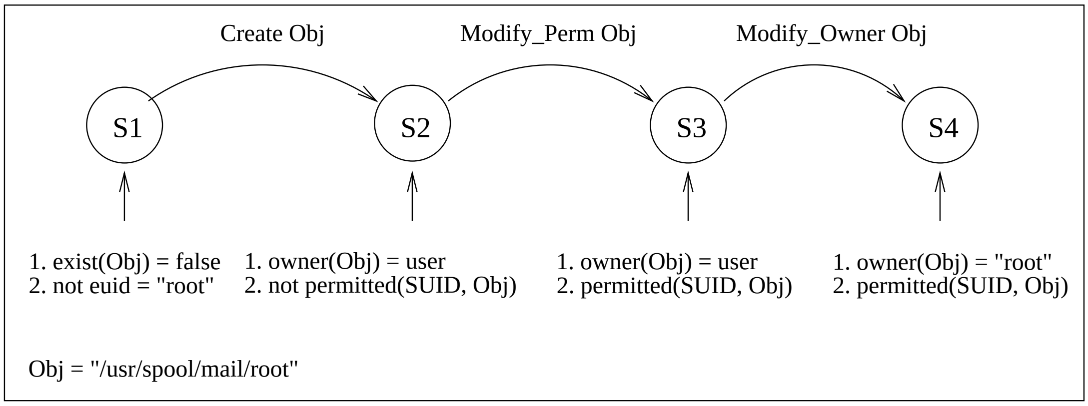

Figure 2.1: State Transition Diagram in STAT.

state back to some previous state, the assertions of which still hold. The rest of state assertions and signature actions depicted in the diagram are self-explanatory and follow directly from our description of the penetration scenario.

In practice, USTAT does not directly use the graphical representation of state transition diagrams, but rather employs a front-end language for encoding these diagrams. To this purpose, the system uses a text file, called *state description table*, which contains in order, the list of state assertions for each state. To distinguish between several state transition diagrams in the rule base, each group of state assertions belonging to a particular state transition diagram is preceded by a header line identifying that particular diagram among the other diagrams in the rule base. Moreover, the system uses another text file, called *signature action table*, which contains in order, the group of signature actions appearing in each diagram. Similarly, a header line identifies each group of signature actions comprising a particular state transition diagram. The state description table and the signature actions table encoding the state transition diagram of Figure 2.1 are depicted in Figures 2.2 and 2.3, respectively.

The advantages of the STAT approach are:

- State transition diagrams provide an intuitive, high-level, and audit records independent representation of penetration scenarios.

- Permutable transitions allows one to represent partial order of signature actions comprising

```
SD1:
exist("/usr/spool/mail/root") = false & not euid = "root".
owner("/usr/spool/mail/root") = user & not permitted (SUID, "/usr/spool/mail/root").
owner("/usr/spool/mail/root") = user & permitted (SUID, "/usr/spool/mail/root").
owner("/usr/spool/mail/root") = "root" & permitted (SUID, "/usr/spool/mail/root").
```

Figure 2.2: State Description Table

```
SIG1:
create("/usr/spool/mail/root").
modify_perm("/usr/spool/mail/root").
modify_owner("/usr/spool/mail/root").
```

Figure 2.3: Signature Action Table

an attack scenario.

- A state transition diagram uses the smallest possible subset of signature actions that must occur for the penetration to be successful, thereby covering multiple variants of the same penetration which are obtained by adding irrelevant user actions.
- The hardlink information maintained by the system adds to the convenience of penetration scenario representations.

Shortcomings of the STAT approach can be summarized as follows:

- The list of primitive state assertions and signature actions are hand-coded and may not be powerful enough for expressing more elaborate penetration scenarios.
- The evaluation of certain state assertions involving information about system attributes or user privileges not available in audit records require that the inference engine must interact, in real-time, with the target system in order to retrieve the needed information. This includes state assertions that are expressed in terms of file type (e.g., binary, script, link) or file existence. This interaction with the target system (through some system calls such as *stat*(2V), *lstat*(2V), and *readlink*(2V)) could incur a performance degradation in the inference engine.

Preliminary experimental results of USTAT [32] were performed on a Sun SparcStation 1 running SunOS 4.1.1 with its C2 Basic Security Module (BSM). These results showed that it took USTAT 1 minute and 46 seconds elapsed time to analyze an audit trail of 7,343 events using 12 attack scenarios with no other major cpu load on the machine used for the test.

2.6.3 Colored Petri Nets

The approach described in [21] proposes an intrusion classification scheme independent of any underlying computational framework of matching. The approach provides a pattern matching-based computational model in which all categories in the classification are represented and matched. The model of matching consists of:

context representation that allows the matching to correlate various events comprising the intrusion signature,

follows semantics that accommodates the possibility of several intrusion patterns (belonging to perhaps multiple event sources) being intermixed in the same event stream,

specification of actions that provides execution of specified actions when the pattern is matched.

The underlying computational framework uses a variation of Colored Petri Nets (CP-nets) to represent and detect intrusion patterns. In this model, an intrusion is represented as a Colored Petri Net, in which a context is modeled as colors of tokens residing in each state. The matching is driven by the audit trail and is performed by moving tokens progressively from (possibly several) initial states to the final state.

A CP-net can have a set of variables associated with it. The variables are handled through unification, which means that variables can be assigned a value only once. Each token maintains its own local copy of these variables because each token can make its own variable binding as it flows from a start state to the final state. A CP-net also contains a set of directed arcs that connect states to transitions and vice-versa. Each transition is associated with an event type, called its label, which must occur in the input event stream before the transition will fire. Nondeterminism can be specified by labeling more than one transition with the same label. A transition is said to be enabled if each of its input states contain at least one token. Optional expressions, or guards, can be placed at transitions. These expressions permit assignment to the token local variables that flow past the transition. Examples of these expressions include assignment of event data fields to token local variables, evaluation of conditions involving $=$, $<$, or $>$, and calling built-in and user defined functions. `this` is a special operator that is instantiated to the most recent event. Event data can be accessed through the `this` operator. Expressions involving this use the array indexing operator `[]` to refer to data from the current event.

A transition fires if it is enabled and an event of the same type as its label occurs that satisfies the guard at the transition. When a transition fires, the set of consistent tokens are unified to one token, and copies of the unified token are placed in each output state of the transition. The states of a CP-net can also be associated with actions which are performed for each token that reaches the state. This allows the encoding of countermeasures when a partial signature is recognized. In addition to the event pattern to be matched against the audit trail, the user can optionally specify another CP-net, called *negative invariant*, the matching of which nullifies the matching of the CP-net representing the event pattern. Negative invariants are used in cases where one needs to match a pattern such as $p_1 \wedge \bar{p}_2$ which means that the pattern is considered matched if a sequence of events e_1, ..., e_n satisfies p_1 but does not satisfy p_2. In this case, a CP-net used for matching p_1 whereas the negative invariant is used for matching p_2. Note that it is possible to represent a partial order of events in a pattern by allowing a transition to have several input states.

An intrusion detection system called IDIOT (Intrusion Detection In Our Time) has been developed based on this approach [33] .

In order to illustrate the use of this approach, Figure 2.4 depicts a literal graphical representation of the penetration scenario presented in the previous section.

The horizontal path in this diagram represents the activity:

```
> cp /bin/sh /usr/spool/mail/root
> chmod 4755 /usr/spool/mail/root
```

while the diagonal chain encodes the activity `touch x`. (Actually, the execution of the command `touch x` uses the system call *stat*(2V) followed by the system call *utimes*(2V).) The transition labeled t_7 represents a synchronization point where both input states s_6 and s_3 must contain at least one token before the transition will fire. For transition t_7 to fire, there must be at least one token in each of states s_3 and s_6, the enabling pair of tokens (one from state s_3, the other from s_6) must have consistently bound (unifiable) variables, and the unified token and the event of the type `exec` together must evaluate the guard at t_7 to true.

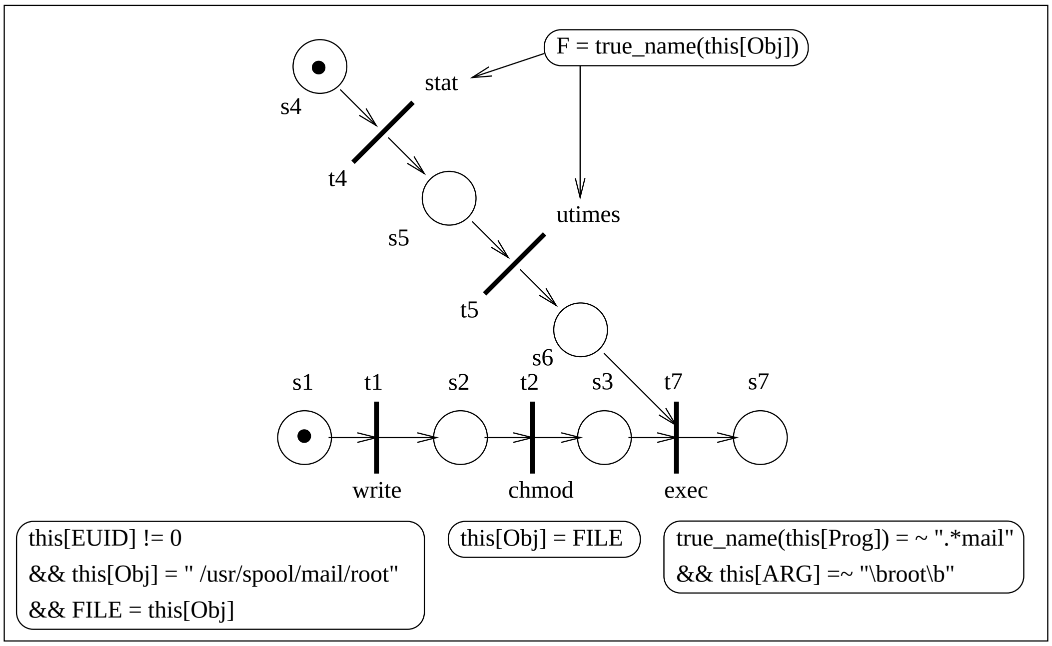

Figure 2.4: Colored-Petri net in IDIOT.

Note that the CP-net of Figure 2.4 only accounts for penetrations that exploit the mail security vulnerability described in the previous section. This means that if the attacker succeeded to change the ownership of the shell copy using some other means, the CP-net of Figure 2.4 will fail to detect it. Consequently, the state transition diagram of Figure 2.1 used by STAT is more general since it detects other variations of the presented scenario. However, the CP-net of Figure 2.5 allows one to detect other variations of this penetration scenario in the same way as in STAT.

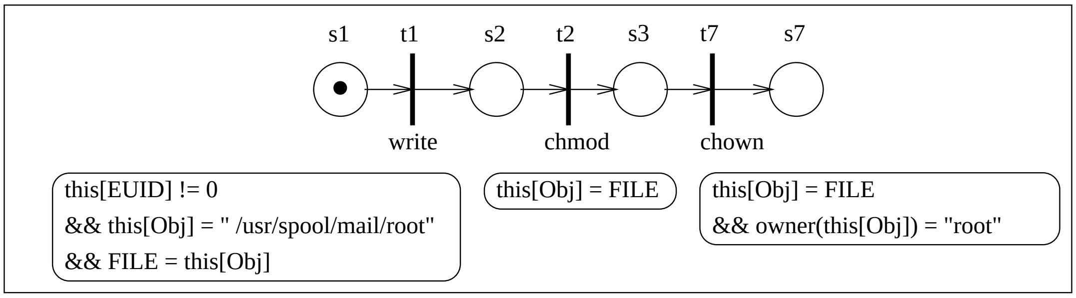

Figure 2.5: Colored-Petri net in IDIOT (second version).

In this CP-net, the diagonal path has been removed since it is not vital for the successful completion of the penetration, and the transition t_7 is labeled by the event type *chown* which changes the ownership of the shell copy to that of root. However, the second version of the CP-net is less efficient in the average since the pattern has to evaluate the truth value of the guard at transition t_7 for all *chown* event type.

```
extern int owner(str);
pattern root_shell "creation of a setuid, root-owned shell copy" priority 7;
   state s1, s2, s3, s7;
   str FILE;
   int EUID;
post_action {
   printf("User %d successfully created a root-owned setuid copy of the shell", EUID);
}
trans t1(WRITE)
   <- s1;
   -> s2;
   |_ {
      EUID = this[EUID] && EUID != 0 && this[Obj] = "/usr/spool/mail/root" &&
      FILE = this[Obj];
   }
end t1;
trans t2(CHMOD)
   <- s2;
   -> s3;
   |_ {
      this[Obj] = FILE;
   }
end t2;
trans t3(CHOWN)
   <- s3;
   -> s7;
   |_ {
      this[Obj] = FILE && owner(this[Obj]) = "root";
   }
end t3;
end root_shell;
```

Figure 2.6: Example Front-end program in IDIOT

Similar to STAT, IDIOT uses a front-end language for encoding the graphical representation of CP-nets. A program in the front-end language is composed of state declarations and CP-net variable declarations. Additionally, each transition specifies an event type (its label), a guard expression, and the input and output states. Finally, an action is specified which is executed upon successful match of the entire pattern. The text file encoding the CP-net of Figure 2.5 is depicted in Figure 2.6.

Timing results of the implemented system show that the system was able to analyze a 400KB audit file corresponding to 2,514 events and using 19 attack scenarios in a 5.91 seconds (accumulated user and system times). The experiment was carried out on a Sun SPARCstation 5 with 32 MB of memory running Solaris 2.3 with its C2 Basic Security Module (BSM) enabled and under light load [21].

2.6.4 Comparing STAT and IDIOT

One major difference between STAT and IDIOT is that in STAT a penetration is detected by the effect it has on the attributes of the system in terms of state changes such as the acquisition by a given user of previously unheld privileges. The audit trail is used for tracking the state changes in the system. By contrast, in IDIOT, a penetration is detected by pattern-matching the audit record sequence, or signature, comprising the penetration. In addition, in STAT, guards are placed in states whereas in IDIOT they are placed in transitions.

While in IDIOT a transitions is triggered by the occurrence of an event of a certain type, in STAT, transitions are triggered by signature actions which can be viewed as abstractions of events. Consequently, several event types may be mapped to the same signature action. For instance, the *Modify_Perm* signature action may correspond to a *chmod* or a *fchmod* system call. Another major difference between the two systems is that in STAT a canonical audit record format similar to that used in the generic intrusion detection model by Denning [8] is used, whereas in IDIOT, audit records are viewed as tuples with a special tag which is the event type. Independence of IDIOT with respect to audit record formats makes it applicable to other problem domains such as the analysis of IP packet streams in a TCP/IP network.

2.7 Problems with Current Intrusion Detection

Although existing anomaly and misuse detection approaches have achieved a degree of success, they still exhibit weaknesses that are primarily related to their models of anomaly and misuse. Practical systems built according to these approaches devise a computational framework for representing and detecting anomalies and/or misuse. As a corollary of the above abstract way of addressing the intrusion detection problem, the computational framework for intrusion detection is often dependent on the model of intrusive behavior. Validating a model of intrusion requires, ideally, a formal proof of its completeness and soundness. The completeness of normal behavior model means that it actually covers all possible instances of normal behavior. Similarly, given a model of misbehavior, to prove its completeness, one needs to show that every instance of a user misbehavior is accounted for in the model. Similar discussion of the soundness problem can be formulated. Additionally, a model of intrusive behavior is inherently tied to a particular site specific security policy. Consequently, as a security policy may both change over time and differ significantly from one site to another, a model of anomaly/misuse should, in principle, be also changed accordingly.

Another shortcoming of the above mentioned abstract approach to anomaly and misuse detection is related to the adequacy of audit data for intrusion detection. Originally, audit trails were designed for accounting and billing purposes, but were not tailored to the requirements of effective security audit trail analysis. As a practical consideration, vendors design auditing mechanisms only to be compliant with the Department of Defense security evaluation criteria class C2 [13] or above, rather than to be effectively useful for an intrusion detection system. Recent audit trails, even though they provide good enhancements over early versions, they are still not adequate for effective audit trail analysis. Deficiencies in current audit data can be described at two different levels.

At the syntactic level, existing audit trail formats are dependent on machine architectures in terms of bit and byte ordering. The disparity in audit data format requires preprocessing in order to ensure the independence of the intrusion detection system with respect to the underlying layout of audit records, and also poses the problem of interoperability of intrusion detection systems. Preprocessing of audit data is only a partial solution because it entails a performance penalty on the monitored system, especially if real-time is a critical requirement.

The semantics level of audit data fields is another area of concern. On the one hand, the list of supported audit data fields — particularly event type field — and their meaning often embodies a security policy, which is again, subject to modification and may not be adequate for the intrusion detection problem. Typically, some elements in the model of normal behavior or misuse behavior cannot always be directly mapped to audit data fields in the audit records. For instance, SMF audit records do not contain a subject field; the subject must be reconstructed by linking together the audit records associated with a given job. On the other hand, although the subject field is supported in many audit trails, its type and representation are not the same. The list of audit record fields is also dependent on the design of operating system's services and interfaces. Standardizing audit record fields at the semantics level hence involves reconciling the design of operating systems themselves. This reconciliation is very difficult to achieve in practice. Even if an accurate model of intrusions is attained, deficiencies in audit data for monitoring user activities might make its implementation infeasible. If, for instance, a statistical profile is used as a model of normal behavior, where a great amount of CPU resource is indicative of a denial of service, if the audit records do not include a CPU usage field, the intrusion detector will be unable to implement this model of intrusion.

Based on the above arguments, the design of a canonical format for audit records does not fully address the problem of portability and reusability of detection rules as part of an intrusion detection system.

2.8 History of ASAX

Historically, ASAX was developed as an audit trail analysis system for the auditing facilities of BS2000 and SINIX, two radically different operating systems from Siemens-Nixdorf A.G. What is worth mentioning is that ASAX's design and implementation were carried out in parallel with that of BS2000 and SINIX auditing facilities, and hence, at the time ASAX was being developed, little knowledge was available about audit record semantic information and format layout. Based on this fact, ASAX's design was driven by two major goals. First, the design of a highly flexible canonical record format, to which operating system-specific audit records can be straightforwardly transformed. Second, the design of an efficient, powerful, yet convenient rule-based language, which is suitable for expressing and detecting arbitrarily complex patterns of audit records. Whereas the design of the auditing facilities was performed by Siemens-Nixdorf, the conception and implementation of the rule-based language was done at the Institut d'Informatique of the University of Namur.

Chapter 3

Rationales

3.1 Problems related to intrusion detection

In this section, we examine some of the main important issues underlying the problem of effective audit trail analysis. These issues serve as guidelines for designing the architecture of our intrusion detection tool.

3.1.1 Complexity and disparity of attack scenarios

The examination of intrusions reported to date shows that penetration scenarios are both complex and diverse [34, 35, 36]. The complexity of penetration scenarios can be enumerated as follows:

Dependency with respect to operating systems: Although penetration scenarios share some commonalities across different operating systems, they are differently manifested in each operating system's audit trail and consequently, an intrusion detection system is likely to employ different detection rules for each operating system. Moreover, the list of security vulnerabilities can be substantially different for each specific operating system. Consequently, the corresponding rule base that is utilized by the intrusion detection system is also likely to be operating system dependent. However, for certain penetration scenarios, such as failed login attempts, the detection rules can be reused with slight variations.

Cooperative attacks: Certain attack scenarios may be perpetrated by two or more users working in concert; each sub-pattern in the attack being performed by a different user. Intrusion detection approaches that focus their analysis on the behavior of each subject taken separately from the other subjects are not effective in handling cooperative attacks.

Contextual attacks: This type of attacks covers most known attack scenarios. They do not consist of a single action but rather of a sequence of related actions. Often, considering each action in the scenario in isolation by means of a record-wise selection of audit events cannot reveal the occurrence of the intrusion or anomaly. Not only these events, or actions, must be detected, but also a context must be maintained in order to be able to correlate the events comprising the signature. A context may consist of the user or process identity to which the perpetrated attack is attributable. Another example is the necessity of remembering the file descriptor of an open file in order to track subsequent I/O operations involving the same physical file.

Across session attacks: These are attack scenarios that span more than a single user session. A user session is defined as the user's activity which takes place between two successive login and logout. In this type of attacks, parts of the penetration pattern are performed within several different sessions related to the same user. Consequently, session-oriented intrusion detection may not be effective in dealing with such attacks. Some intrusion detection systems such as HAYSTACK [24] and MIDAS [26] update user profiles and evaluate intrusion types upon session termination.

Distributed intrusions: The problem of audit trail analysis and intrusion detection is further complicated in a distributed environment wherein parts of an attack signature do not involve a single audit trail, but rather multiple audit trails that are collected at various hosts with possibly different architectures, operating systems, and auditing mechanisms. This issue is addressed in Chapter 10.

3.1.2 Size of the audit files and the selection problem

A challenging problem related to efficient audit trail analysis is the huge size of audit trails. Current auditing mechanisms can collect hundreds of megabytes of audit data each day, even for modest sized user communities. These mountains of audit data place a heavy burden on the I/O and CPU performance, and also makes efficient audit data analysis difficult to achieve in practice. Controlling the amount of audit data can be addressed at two different levels. Audit data collection facilities of modern operating systems provide the security officer with the ability to control the amount of generated audit data at the source level, whereby the security officer specifies which event classes must be recorded in the audit trail. Source level control of audit data volume is referred to as *pre-selection*. Audit data pre-selection can also be specified based on the outcome (success or failure) of audited actions. The pre-selection must be carefully specified, otherwise auditable events that are critical for effective investigation, intrusion detection, and damage assessment may be missed, making the audit trail unusable. Consequently, the security officer must know exactly what s/he is looking for beforehand.

The other level of audit data volume control, called *post-selection*, is performed at the analysis level and consists of selecting relevant audit events from the audit data repository. In practice, a post-selection phase makes use of simple selection rules and has the advantage of relieving the audit trail analyzer tool of a lot of irrelevant data. This is the case for the systems MIDAS [26] and IDES [8, 11, 12, 28, 37].

Clearly, in determining the amount of collected audit data, there are several tradeoffs to consider, including the I/O and CPU resources available, the complexity and types of intrusions that can be detected, the level of security of the target system, and the ability to perform damage assessment.

3.1.3 Disparity of the Audit data

Some of the issues related to the adequacy of audit data to the problem of audit trail analysis were outlined in chapter 2. Other specific issues are discussed here. What is apparent from many situations is that the necessary audit data that is critical for effective detection and identification of attack scenarios are simply missing in the audit trail. For instance, audit records do not incorporate command line arguments, environment variables such as the search path `PATH` or the `IFS` variable (which indicates which characters are to be treated as blanks by the shell), or dynamically linked libraries. These items are necessary for the attack pattern to be correctly matched. For example, it is not possible to figure out which program a command refers to if the latter is invoked through its base name and if the environment variable `PATH` is not recorded in the audit trail. Moreover, in SunOS 4.1.1 BSM, the *chmod* system call only provides the new permission modes of the file and not the old ones. Another example concerning Solaris 2.x BSM is that a reference to a symbolic link file is recorded as the reference to the target of the link instead of the link itself. If several links are made to the same physical file, it is not possible to tell which symbolic link was actually referenced.

3.1.4 Real-time Audit Data Analysis

Audit trail analysis, when possible, can be conducted in real-time, i.e., audit records are analyzed as they are generated by the auditing subsystem. This allows the security officer to take timely countermeasures before extensive damage is perpetrated on the system. Real-time analysis is often performed in highly security-sensitive environments, in which the destruction or disclosure of secret data cannot be afforded. The feasibility of real-time analysis is mainly determined by the rate of audit data generation, the complexity of intrusions to be detected, and the efficiency of the audit data collection mechanism itself.

The real-time analysis also provides the opportunity of real-time surveillance, whereby an increased auditing granularity level is dynamically performed for those users who exhibit a suspicious behavior. The surveillance capability can also be done statically for a predetermined list of misbehaving users. Dynamic surveillance is however a more desirable goal since it does not require manual specification of suspected users ahead of time. The pre-emption of attacks is highly desirable because it allows the security administrator to abort an intrusive activity on the fly, before it is successfully completed, or at least, before extensive damage is done.

3.2 Requirements for an Intrusion Detection System

In this section, we identify the requirements for an intrusion detection system able to overcome the problems mentioned in Section 3.1.

3.2.1 Power and Convenience

Due to the complexity of penetration scenarios, a major requirement of an intrusion detection system is that it incorporates a powerful computational model for representing and detecting intrusion patterns. Additionally, the system must provide a convenient way to express and detect attack signatures. Whereas general purpose languages such as C or Pascal are powerful enough, they are not sufficiently specific for the purpose of audit trail analysis. In order to be convenient, the computational model of matching attack signatures against the audit trail must support built-in ready-to-use features and control structures that are adequate and tailored for the specific needs of audit trail analysis. User convenience also means an easy and intuitive way of expressing the intrusion patterns to be detected. Since it is not feasible to enumerate exhaustively all present and future attack scenarios, *Ad-hoc* approaches are ruled out because they provide only a limited expressive power. In such approaches a new detection capability must repeatedly be added to the detection engine whenever a new attack scenario happens not to be describable by the available capabilities.

3.2.2 Reusability

A worthy quality of a software is its ability to be reused in different environments with a minimum adaptation effort. In the software engineering literature, reusability is addressed at different levels of software development. This includes reusability of code, detailed design, global design, or even specification components. Audit trail analysis tools are no exception to this required quality. It is worth being able to reuse an audit trail analysis tool, or at least the knowledge encapsulated within it, when the environment changes. Such changes may involve the machine architecture, the operating system, the auditing mechanism, the security policy, or any combination of these.

Reusability of audit trail analysis tools can be considered either at the logical level or at the physical level. Reusability at the logical level is well exemplified by the generic intrusion detection model [8] underlying many intrusion detection systems built to date. This model provides a general framework for describing intrusion detection systems in terms of abstract components: *subjects*, *objects*, *profiles*, *activity rules*, etc. The model eases not only the portability of the logical design of a system, but also the comparison between different systems.

3.2.3 Efficiency

This requirement is critical for intrusion detection. First, the complexity of attack scenarios requires an efficient detection engine that is capable of pattern-matching complex penetration scenarios as manifested in the audit trail. Second the real-time requirement assumes an efficient analysis engine that is able to analyze audit data at a rate faster than audit data generation. Third, the huge size of audit files makes the efficiency requirement an important factor even for off-line analysis.

3.2.4 Adaptability

Different sites may follow different security policies driven by the sensitivity of information. The more security-sensitive is the information, the more likely that the security policy is tightened. Typically, a user behavior may be considered benign in some sites whereas the same behavior could be viewed as highly anomalous or dangerous in other sites. Correspondingly, different security administrators may be interested in monitoring or investigating different behavior patterns depending on their respective site security policies.

The adaptability requirement means that the intrusion detection tool should be easily customizable in order to meet specific security policies. According to this goal, the security administrator should be able to selectively activate detection rules that are relevant to his/her security concerns and similarly, s/he may omit the activation of detection rules that are not relevant. Alternatively, s/he may possibly incorporate new detection rules that are better suited to watch for what s/he is specifically looking for.

3.3 Our Approach

In the present section, we describe the salient features of the language RUSSEL that we use as a computational model for analyzing data streams in general and security audit trails in particular. While RUSSEL was developed for powerful, universal, and efficient audit trail analysis, its features make it general enough to be applicable to the analysis of any data stream. We however limit ourselves to demonstrating these features for the specific audit trail analysis problem; in [38], we demonstrate its applicability to virus detection and classification. In the following, we describe each feature and show how these features allow us to fulfill the requirements presented in the previous section.

3.3.1 Bottom-up Rule-based Language

RUSSEL is a rule-based language specifically designed for audit trail analysis. The idea is to provide a language having the power of a general purpose rule-based language but which is specifically tailored for processing large sequential files efficiently. The convenience of rule-based languages is generally recognized by the intrusion detection community: the intrusion detection system IDES [6] uses a general rule-based expert system P-BEST (Production-Based Expert System Tool), while the OSIRIS system [29] uses Prolog rules.

RUSSEL supports guarded actions of the form *Condition* `-->` *Action*, which are convenient for sequential audit trail analysis: given that a security-related event satisfies the *Condition*, the security officer wishes to perform a specific *Action*. The latter could be an alarm report, a filtering, or some kind of further analysis, such as correlating the current event with subsequent events in the audit trail. The idea of RUSSEL is to take advantage of such particular utilization to make the language as efficient and easy to use as possible.

The language is bottom-up because it starts with the collection of audit records considered as basic facts, and based on these facts, it tries to find out arbitrarily complex audit record patterns which can be viewed as derived facts, or goals. The bottom-up search strategy is utilized because, on the one hand, we are analyzing a sequence of audit records which is not entirely known beforehand (i.e., at the time the analysis starts), especially in case the analysis is on-line. On the other hand, a top-down search strategy as provided by declarative languages such as Prolog, is not efficient enough for the analysis of huge audit trails, because the search engine must perform a backward chaining in exploring the search space (i.e., it should backtrack on the audit records comprising the audit trail). Given the overwhelming amount of audit data to be analyzed, such a top-down analysis is likely to be impractical at least if efficiency is a main concern.

3.3.2 Imperative Style

In order to ensure a good efficiency, the rule-based language RUSSEL exhibits an imperative programming style, as opposed to a declarative one. In RUSSEL, a *rule declaration* is made of a name, a set of formal parameters, a set of local variables, and an action part. The latter uses classical control structures such as composition of actions, conditional actions, and repetitive actions. Elementary actions include assignment, built-in and user-defined routine calls, and *rule triggering*. Rule triggering is a mechanism whereby a rule is made active for the current record, the next record, or at termination of the analysis of the whole audit file. An *active* rule is the instantiation of a rule declaration with a list of actual parameter values.

The overall processing of the audit trail could be roughly summarized as follows:

- The audit trail is analyzed sequentially record by record by means of a collection of analysis rules. At a given time, there is a current record being analyzed and a collection of active rules.

- Active rules encapsulate all the relevant knowledge about the past analysis. This knowledge is then applied to the current record by executing these active rules against it.

- Triggering a rule off involves supplying the actual parameter list of the rule and specifying whether the rule has to be triggered off for the current record, for the next record, or at the end of the analysis of the whole audit file.

- The analysis process is initialized by the so-called *init action* which specifies a set of rules to be activated for the first record in the audit trail.

Therefore, specifying an audit trail analysis in RUSSEL consists of writing a number of adequate rules which will be activated according to the above global control scheme. From a programmer viewpoint, a rule is like a classical parameterized procedure, which may additionally involve a particular kind of actions: rule triggering.

Thus RUSSEL can be viewed as an imperative language including a particular *predefined control structure* suitable to make reasoning about sequences of records.

Support of contextual attacks is naturally provided in RUSSEL through the built-in rule triggering mechanism which allows an active rule to trigger off another rule with a list of parameters that encapsulate the relevant data items comprising the context. Assuming for instance one is interested in detecting all activity records related to a particular process, a first rule detects the audit record that reports a process initiation and triggers off a second rule with that process ID as a parameter in order to match it against subsequent records. Such a rule can also be passed the user ID in order watch for all activity data accountable to that user ID, thereby providing user surveillance. Moreover, if across session monitoring is desired, for example in case only a part of a penetration scenario has been detected during the current session, such a rule can stay active after session termination (when encountering a logout event) in order to watch for the completion of the penetration during subsequent sessions.

3.3.3 Normalized Audit Data Format

Reusability in ASAX is addressed at the physical level. The aim is to provide a universal audit trail analysis tool which runs on a large spectrum of operating systems. The rationale of this choice is twofold: i/ to make it possible to analyze any audit trail provided this audit trail has been previously translated to an appropriate format, ii/ to ensure the maximum efficiency which remains the major concern in audit trail analysis.

Practically, we defined a normalized audit trail format (NADF) which is flexible enough to ensure that existing audit trails can be translated to it in a straightforward way. Audit trail analysis is then performed on normalized audit trails only. So-called *format adaptors* should be provided to translate each native audit trail to NADF format. Feasibility of the approach has been demonstrated by providing format adaptors for existing audit trails on BS2000, SINIX (two operating systems from Siemens), SunOS 4.1.1, and SunOS 5.x.

The translation is relatively straightforward because the NADF format is extremely simple and flexible. A NADF file actually is a chronological sequence of records, where a record consists of a list of audit data each of which being composed of an audit data identifier and an audit data value (there is no type!).

In addition, a format adaptor may generate other auxiliary files that conserve the correspondence between external and internal formats of audit data: association tables, decoding routines, etc. This allows analysis queries to be expressed using the external format. A low level description of the NADF files is given in Section 6.2 (for UNIX systems). The low level description is irrelevant to the writing of RUSSEL programs but it must be known to design correct format adaptors.

Moreover, in order to relieve security administrators from the task of implementing format adaptor programs for translating native audit trails to the NADF format, a format adaptor generator tool is provided that automates this task for a variety of audit record structures.[1]

3.3.4 Modularity

The language supports modules which share global variables and external rules whereby taking advantage of the classical modular design methodology including information hiding, data sharing, incremental development, and reusability. Every RUSSEL module can be seen as a collection of detection rules and global variables which implement the detection of a particular penetration scenario. Only top-level rules (and possibly some global variables) are made visible to other modules in the rule base. RUSSEL rule base can then be viewed as a collection of modules that provide the security administrator with the ability of selectively choosing the set of modules that are of interest to the local site security policy without introducing unexpected interferences between modules. The modularity of the language hence addresses the adaptability requirement.

3.3.5 Minimality and Interfacing to a General Purpose Language

In order not to clutter the language with features that have nothing to do with the sequential analysis of audit trails, only a limited yet extensible set of predefined actions are provided by the language. However, a clean and well defined interface to the C language is supported that allows a programmer to use procedures for reading input, writing output, managing complex data structures and interacting with the target environment. In particular, the C-interface can be used for:

- notifying the security administrator, perhaps in real-time, through an e-mail message describing the occurrence of a suspicious activity, together with the identity of involved parties (user ID, times-tamp, file names, host name, etc.). The latter items describing the activity are taken from the active rule's parameters;

- pre-empting intrusive behavior by terminating an intrusive process, a user session, or suspending a user's account altogether;

- dynamically altering the granularity level of a user if the current rules detect a suspicious activity. This can easily be done by a C-routine that interfaces to the audit daemon to modify the set of event classes being collected for that user.

3.3.6 Careful and Portable Implementation

Efficiency and adaptability are two important requirements which must be supported by a careful implementation of the language. Adaptability is supported by the use of an intermediate internal code, which allows for minimal reprogramming when porting the system to a new platform. Efficiency is addressed by paying particular attention to the implementation of the most critical constructs of the language RUSSEL, i.e., conditions. As a matter of fact, most rules basically consist of complex conditions about the current record which trigger comparatively simple actions. Thus, it is important to ensure that conditions are evaluated as efficiently as possible. This notably requires to have a fast access (in constant time) to the fields of the current record in spite of the

[1]The exact specification of the format adaptor generator tool together with sample applications will be given in the second version of this thesis.

completely variable format of NADF records. The details of our solution to these problems are given in Section 6.7.

Chapter 4

Informal Overview of RUSSEL

The concept of module supported in the language corresponds to a source file containing a collection of rule and global variable declarations. Some of these variables and rules can be declared internal; in this case they are only visible to rules that are declared in the same source file, or external; in this case they are visible to all other modules. The declaration of an internal (resp. external) global variable or rule is preceded by the keyword **internal** (resp. **external**). A large source file implementing a complex analysis can consequently be broken into several smaller source files each of which being a module. Each of these modules implements a particular analysis and interfaces to other modules through the declaration of external global variables and/or external rules. Every module can optionally declare its own, or private, init action which is internal by definition. In certain cases, it is convenient to gather all global variables that are shared (i.e., declared external) by all modules within a single module. The latter may have its own init action that initializes these shared variables.

Additionally, the language provides a mechanism that allows a module to use another module. The latter module can use still another one and so on. This can introduce loops in the chain of module usage and this is well permitted by the compiler. The latter maintains a list of already compiled modules in order not to loop for ever compiling modules. The set of all these modules makes up the whole program that is eventually executed by the language RUSSEL interpreter.

A rule declaration is composed of a rule name, a declaration of a list of formal parameters, a declaration of a list of variables, and an action part, which is also referred to as the rule body. A module must declare an external global variable if any of its rules references this variable, whereas a module need not declare an external rule if any of its rules references it (i.e., if the module contains at least one rule declaration that triggers off that external rule). The reason behind this is that the attributes of a rule triggering (arity, argument types, ... etc) are determined by examining this very rule triggering while nothing (especially the type) can be deduced from a reference to a global variable. However, other choices would have been possible; for instance we would have required a kind of prototype (as in C) of an external rule before it can be referenced.

The execution of a RUSSEL program involves the sequential analysis of an audit trail, one record at a time. The audit record being analyzed at a given time is referred to as *the current audit record*. The latter can be referenced in a rule body only through the names of its fields, such as `USER_ID` or `TIMESTAMP`. So field names are actually treated as implicitly declared global variables names in the RUSSEL program. Note that there is no notion of record type in RUSSEL. Instead, any possible field can be either present or absent in the current record depending on the nature of the particular event represented by the record. Consider the set of all field names that may potentially be present in an audit record. Reading the next audit record has the effect of assigning a value to those fields that are present in the record while leaving the other fields undefined. However, it is possible to test whether a field is present or not.

A rule instance of a rule declaration (also called an active rule) is a pair composed of a rule declaration and a list of actual parameter values. Every instance of a rule is a member of one of the three sets of rule instances: *i/ Current*: the set of rule instances that have already been triggered for the current record and not executed yet; *ii/ Next*: the set of rule instances that have been triggered for the next record; *iii/ Completion*: the set of all rule instances that have been triggered to be executed at completion of the whole audit file analysis. The processing of the current record consists of the execution of all rule instances in the set *Current*, one at a time. The rule instance that is currently executed against the current record is referred to as the currently executed rule instance or simply the *current rule*. The execution of the active rule may possibly modify global variables[1] and update the sets *Current*, *Next*, or *Completion*.

The use of the language RUSSEL for audit trail analysis is better introduced by a typical example: detecting repeated failed login attempts from a single user during a specified time period. This example uses the SunOS 4.1 auditing mechanism. Native audit trails are first translated to the standard NADF format. Hence, the description below is based on the NADF format of the records.

Assuming that login events are pre-selected for auditing, every time a user attempts to log in, an audit record describing this event is written into the audit trail. Audit record fields (or audit data) found in a failed login record include the time stamp (AU_TIME), the user id (AU_TEXT_3) and a field indicating success or failure of the attempted login (AU_TEXT_4). Notice that audit records representing login events are not necessarily consecutive since other audit records can be inserted for other events generated by other users of the system. In the example (see Figure 4.1), RUSSEL keywords are noted in bold face characters, words in upper case identify fields in the current audit record, while rule parameters are noted in italic. This convention will be used throughout this thesis. Two rules are needed to detect a sequence of failed logins. The first one (*failed_login*) detects the first occurrence of a login failure. If this record is found, this rule triggers off the rule *count_rule* which remains active until it detects *count_down* failed logins among the subsequent records or until its expiration time arrives. The parameter *suspect_uid* of rule *count_rule* is needed to count only failed logins that are issued by the same user on a given (*target_uid*). If the current audit record does not correspond to a login attempt from the same user, *count_rule* simply retriggers itself for the next record otherwise. If the user id in the current record is the same as its argument and the time stamp is lower than the expiration argument, it retriggers itself for the next record after decrementing the count down argument. If the latter drops to zero, *count_rule* writes an alarm message to the screen indicating that a given user has performed *maxtimes* unsuccessful logins within the period of time *duration* seconds. In addition, *count_rule* retriggers the *failed_login* rule in order to search for other similar patterns in the rest of the audit trail.

In order to initialize the analysis process, the special rule **init_action** makes the *failed_login* rule active for the first record and also makes the *print_results* rule active at completion of the analysis. The latter rule is used to print results accumulated during the analysis such as the total number of detected sequences.

Evaluation of boolean conditions is optimized as much as possible. However, it is not desirable for the user to be aware of the exact optimizations because this would forbid all kinds of further

[1]Actually, a rule is not permitted to modify the parameters of the rule nor is it permitted to alter the current record content. The reason for this restriction is explained in section 5.6.2

```
global v: integer;

rule failed_login(max_times, duration: integer);
if        AU_EVENT = 'login_logout'
  and AU_TEXT_4 = 'incorrect password'
  --> trigger off for_next count_rule(AU_AUID, AU_TEXT_3, strToInt(AU_TIME)+duration, max_times-1)
fi;

rule count_rule(suspect_auid, target_uid: string; expiration, count_down: integer);
if         AU_AUID = suspect_auid
  and    AU_EVENT = 'login_logout'
  and AU_TEXT_4 = 'incorrect password'
  and AU_TEXT_3 = target_uid
  and strToInt(AU_TIME) < expiration
  --> if count_down > 1
        --> trigger off for_next count_rule(suspect_auid, target_uid, expiration, count_down-1);
        count_down = 1
        --> begin
              v := v + 1;
              println(gettime(AU_TIME), ':  3 FAILED LOGINS ON ', target_uid, ' BY ', suspect_auid);
              trigger off for_next failed_login(3,120)
            end
      fi;
  strToInt(AU_TIME) > expiration
  --> trigger off for_next failed_login(3,120);
  true
  --> trigger off for_next count_rule(suspect_auid, target_uid, expiration, count_down)
fi;

rule print_results;
begin
  println(v, ' sequence(s) of bad logins found')
end;

init_action;
begin
  v := 0;
  trigger off for_next failed_login(3, 120);
  trigger off at_completion print_results
end.
```

Figure 4.1: RUSSEL module for failed login detection on SunOS 4.1

changes and improvements to the system. Therefore, we give evaluation rules that are compatible with all possible optimizations without being committed to any particular one.

Let *cond* be a condition of the form:

$$cond_1 \ \omega \ cond_2 \ (\omega \in \{ \ \textbf{or}, \textbf{and}\})$$

cond will evaluate to the boolean value $b_1 \ \omega \ b_2$ if (but not only if) $cond_1$ and $cond_2$ evaluates to b_1 and b_2 respectively. However, this does not imply that both subconditions will actually be evaluated because the value of $b_1 \ \omega \ b_2$ could depend on only one of the two values. (For example, **true or** b_2 is **true** independently of b_2 and b_1 **and false** is **false** not taking care of b_1).
So the implemented system will be obliged to return $b_1 \ \omega \ b_2$ each time the actual evaluation returns b_1 and b_2 for $cond_1$ and $cond_2$ respectively. But it will be free to do it **without** evaluating some sub-conditions.
Note also that the value of *cond* will sometimes be defined while the value of a sub-condition is not. For example, suppose that *foo* is a field name not present in the current record, then the condition

$$\textbf{true or } foo = \text{'TOTO'}$$

will evaluate to **true** although the value of

$$foo = \text{'TOTO'}$$

is undefined. This feature will prove to be useful in practice because it provides a clean way to process the audit record whose most fields are optional.
Finally, the evaluation of *cond* could well properly terminate in practice although an execution error is possible according to the semantic rules. For example,

$$\textbf{true or } (1 \textbf{ div } 0 = 0)$$

can (but must not) evaluate to **true**. However, as this is only a possibility, the user is not allowed to use such bizarre constructs (that could be evaluated differently in different situations or in different versions of the system).

Chapter 5

Syntax and Operational Semantics of RUSSEL

5.1 Introduction

The present chapter provides a complete and rigorous description of the language RUSSEL in a machine-independent way by abstracting the target system-dependent information as much as possible. In Section 5.2, we define the data types supported in the language. The syntax of rule declaration and modules presented in Section 5.3 is a pre-requisite for understanding the rest of this chapter. Section 5.4 provides some preliminary definitions that are used throughout description of the operational semantics. Section 5.5 contains the semantics of expressions. Section 5.6 contains the semantics of action execution. In Section 5.7, we provide the semantics of processing the current record. In Section 5.8, we provide the semantics of processing a sequence of records. Finally the semantics of executing a RUSSEL program on an audit file is provided in Section 5.9.

5.2 Data types

RUSSEL uses only two simple data types namely **string** and **integer**. The **string** data type denotes arbitrary strings of bytes, and the **integer** data type denotes signed four bytes integer values, i.e., the values in the range $[2^{31}, 2^{31} - 1]$. The basic primitive operations on the **string** type are $<$, $\leq$, $>$, $\geq$, $=$, and !=. The **integer** type supports common primitive operations[1] which correspond to the operators $<$, $\leq$, $>$, $\geq$, $=$, !=, $+$, $*$, **mod**, and **div**. We moreover use a special value, denoted by **undef**, to refer to an undefined value.

Supporting only these two data types has the benefit of keeping the language as simple as possible while allowing it to be useful in real life applications. In fact, the **string** type allows one to represent any data structure by considering it as uninterpreted string of bytes. Thus, the user has the ability to manipulate such a data structure through the use of the C-interface which can be employed to simulate operations on the desired data structure. For instance, a two dimensional array could be simulated by a C-routine that allocates a storage for it and also by various other routines that access and update entries in the array.

[1]Additional built-in operations in fact are available and will be described later on. Moreover, the user has the ability of defining new operations as specified in Section 6.5

5.3 Variables, Rules, and Modules (Syntactic Aspects only)

5.3.1 Global Variable declaration

The BNF syntax of a global variable declaration part is:

⟨global var decl part ⟩ ::= ⟨empty ⟩ |
⟨global var group ⟩ ; ... ; ⟨global var group ⟩ ;

⟨global var group ⟩ ::= ⟨class ⟩ **global** ⟨variable group ⟩

⟨class ⟩ ::= ⟨empty ⟩
|**external**
| **internal**

⟨variable group ⟩ ::= ⟨variable name ⟩ **, ...** , ⟨variable name ⟩ **:** ⟨type ⟩

⟨variable name ⟩ ::= identifier

⟨type ⟩ ::= **integer**
| **string**

⟨empty ⟩ ::=

An empty class is considered equivalent to **external**. A global variable name appearing in a variable group preceded by the keyword **internal** (resp. **external**) is said to be an internal (resp. external) global variable. All global variables of a single module must have distinct names.

5.3.2 Rule declaration

The BNF syntax of a rule declaration is as follows.

⟨rule declaration ⟩ ::= ⟨rule heading ⟩ ⟨formal parameter declarations ⟩
⟨local variable declarations ⟩ ⟨action ⟩

⟨rule heading ⟩ ::= ⟨class ⟩ **rule** ⟨rule name ⟩

⟨formal parameter declarations ⟩ ::= ⟨empty ⟩
| **(** ⟨parameter list ⟩**)**

⟨rule name ⟩ ::= ⟨identifier ⟩

⟨parameter list ⟩ ::= ⟨variable group ⟩ **;** **...** **;** ⟨variable group ⟩

⟨local variable declarations ⟩ ::= ⟨empty ⟩
| **var** ⟨variable group ⟩ **; ... ;** ⟨variable group ⟩

Formal parameters and local variables appearing in a rule declaration must all have distinct names. If the class is empty, the declaration is considered as if the class is the keyword "internal". When a rule declaration is preceded by the keyword **internal** (resp. **external**) the rule is said to be an internal (resp. external) rule.

5.3.3 Modules

A *program* in RUSSEL is a sequence of one or more text files, called *modules*, each of which being composed of four parts: the module usage part, the global variable declarations part, the rule declarations part, and an init action. The BNF syntax of a module is as follows.

$\langle$module $\rangle$ ::= $\langle$usage part $\rangle$
$\langle$global var decl part $\rangle$
$\langle$rule declarations part $\rangle$
$\langle$init action $\rangle$ **.**

$\langle$usage part $\rangle$ ::= $\langle$empty $\rangle$ |
$\langle$module usage $\rangle$ **; ... ;** $\langle$module usage $\rangle$ **;**

$\langle$module usage $\rangle$ ::= **uses** $\langle$module name $\rangle$ **, ... ,** $\langle$module name $\rangle$

$\langle$module name $\rangle$ ::= identifier

$\langle$rule declarations part $\rangle$ ::= $\langle$empty $\rangle$ |
$\langle$rule declaration $\rangle$ **; ... ;** $\langle$rule declaration $\rangle$

$\langle$init action $\rangle$ ::= **init_action ;**
$\langle$local variable declarations $\rangle$ $\langle$action $\rangle$

The first module in the sequence is referred to as the *main module*. The sequence of modules comprising a program are identified by their names M_1, ..., M_n and additionally, the modules must satisfy the conditions:

1. $\forall i \in \{2, ..., n\}\ \exists j \in \{1, ..., n\}\ i \neq j \Rightarrow M_j\ uses\ M_i$
2. $\forall i \in \{1, ..., n\}\ uses(M_i)\ \subseteq \{M_1, ..., M_n\}$

where we denote by $uses(M_i)$ the set of module names appearing in the usage part of a given module M_i, and M_i *uses* M_j means $M_j \in uses(M_i)$.

In other words, every module other than the main module must be used by at least one module other than itself, and all modules used by a given module must also be members of the sequence of modules.

Additionally, all internal rules of a given module must have distinct names and the external rules of all modules must all have distinct names. However, an internal rule may have the same name as an external rule of another module.

5.4 Basic Notions for the Operational Semantics

This section defines basic notions and notations that will be used throughout the description of the language RUSSEL.

Definition 1 Let $V = \{name_1, ..., name_n\}$ be a set of variables names, a *variable assignment* over the set V is a set of the form

$$\mathrm{B} = \{(name_1,\ v_1),\ ...,\ (name_n,\ v_n)\}$$

where v_i $(1 \leq i \leq n)$ is a value of the type of the variable $name_i$ or is the undefined value **undef**; we denote by $\mathcal{N}(\mathrm{B})$ the set of variable names V.

Let us denote by F the set of all field names which may possibly be contained in any audit record. Since any possible audit record field can either be present or absent in an audit record depending on the particular event represented by the audit record, we formally define a record as a variable assignments over some *subset* E of F. Given a record and a field name $fname \in F$, *fname* is said to be present in the given record if the subset E contains a variable assignment for *fname*; it is said absent otherwise.

Definition 2 A *record* is a variable assignment over a set of variables, called *fields*, where all variables are of the type string.

Definition 3 Given a rule declaration (or an init action declaration) R, a *rule instance* corresponding to the rule R is a pair $(R,\ B)$, where B is a variable assignment over the set of formal parameters appearing in R.

An expression appearing in a given rule body is evaluated with respect to a hierarchical context which is composed of the local variables (V_4), the formal parameters (V_3), the global variables (V_2), and the current record (V_1).

Definition 4 An *expression evaluation context* is defined as a 4-tuple $\mathcal{E} = (\mathrm{B}_1,\ \mathrm{B}_2,\ \mathrm{B}_3,\ \mathrm{B}_4)$ where B_1, B_2, B_3, and B_4 are variable assignments over some sets of variables names V_1, V_2, V_3, and V_4, respectively.

Definition 5 Given an expression evaluation context $\mathcal{C} = (\mathrm{B}_1,\ \mathrm{B}_2,\ \mathrm{B}_3,\ \mathrm{B}_4)$ and a variable name *name*, the variable is said to be *active* in $\mathcal{E}$ if and only if there exists some $i \in \{1, ..., 4\}$ such that

$$name \in \mathcal{N}(\mathrm{B}_i)\ \ \wedge\ \ \forall\, j \in \{\mathrm{i{+}1},\ ...,\ 4\}\ \ name \notin \mathcal{N}(\mathrm{B}_j)$$

The variable is said to be *inactive* otherwise.

From the above definitions, every expression evaluation context contains at most one active variable of a given name, and we say that such a name *designates* that variable. A variable name appearing in an expression may designate either a local variable, a formal parameter, a global variable, or a field name, in that order.

The execution of an action appearing in a given rule declaration is performed with respect to a context composed of the expression evaluation context related to the given rule, the internal rules, the external rules, and the three sets of rule instances: I_1, I_2, and I_3, which are have been triggered for the current record, the next record, and at completion time, respectively.

Definition 6 An *action execution context* is defined as a 9-tuple $\mathcal{A}$ = (B_1, B_2, B_3, B_4, $\mathcal{R}_1$, $\mathcal{R}_2$, I_1, I_2, I_3) where

- (B_1, B_2, B_3, B_4) is an expression evaluation context,
- $\mathcal{R}_1$ and $\mathcal{R}_2$ are sets of rule declarations,
- I_1, I_2, and I_3 are multi-sets of rule instances.

Definition 7 Given an action execution context $\mathcal{A}$ = (B_1, B_2, B_3, B_4, $\mathcal{R}_1$, $\mathcal{R}_2$, I_1, I_2, I_3) and a rule declaration R, the rule is said to be *active* in $\mathcal{A}$ if and only if R $\in \mathcal{R}_1 \cup \mathcal{R}_2$. The rule is said to be *inactive* otherwise.

From the above definitions, every action execution context contains at most one active rule of a given name, and we say that such a name *designates* that rule. Consequently, the rule name appearing in a rule triggering designates either an internal rule or an external rule, in that order. We consider multi-sets of rule instances because several instances of the same rule with same actual parameter values may have been triggered.

The current record is processed with respect to a context containing the current record, the external rules and the three sets of rule instances I_1, I_2, and I_3.

Definition 8 A *record processing context* is defined as a 4-tuple $\mathcal{C}$ = (B, $\mathcal{R}$, I_1, I_2, I_3) where

- B is a variable assignment over some set of variables V;
- $\mathcal{R}$ is a set of rule declarations;
- I_1, I_2, and I_3 are multi-sets of rule instances.

The effect of processing the current record is to modify the global variables, I_1, I_2, or I_3.

For the sake of clarity, we assume that internal and external global variables in any module all have distinct names. If this is not the case, it is always possible to rename these variables so that they all have distinct names. This can be done for instance by prepending the name of global variables by their respective module names.

5.5 Expressions and Conditions

5.5.1 Literals

There are two kinds of literals in RUSSEL: C-literals and X-literals. A C-literal is a quoted string of ASCII printable characters (e.g., 'UNIX'). Two successive quote characters within a C-literal denote a single quote character.

X-literals are introduced in order to denote strings of arbitrary bytes and are needed because audit data fields may contain non printable characters such as control characters, binary representations of numbers, coded values, etc. An arbitrary string is denoted by a sequence consisting of an even number of hexadecimal digits, each pair of digits represents a single byte in the string. The sequence of hexadecimal digits is quoted and is preceded by the letter `X`. Let

$$X'\, X_{1,1}X_{1,2}\, X_{2,1}X_{2,2}\, , \cdots , X_{n,1}X'_{n,2}$$

be an X-literal, where each $X_{i,j}$ denotes an hexadecimal digit. By definition, this literal denotes the string of bytes B_1, ..., B_n, where $B_i = 16 * X_{i,1} + X_{i,2}$. (We identify an hexadecimal digit with its value.)

5.5.2 Arithmetic expressions

Syntax

$\langle$arithmetic expression $\rangle$ ::= $\langle$literal $\rangle$
 | $\langle$variable name $\rangle$
 | $\langle$arithmetic expression $\rangle$ $\langle$binary arithmetic operator $\rangle$ $\langle$arithmetic expression $\rangle$
 | $\langle$unary arithmetic operator $\rangle$ $\langle$arithmetic expression $\rangle$
 | $\langle$predefined function call $\rangle$
 | **(** $\langle$argument list $\rangle$**)**

$\langle$relational operator $\rangle$ ::= $< \mid > \mid \mathbf{!=} \mid = \mid \leq \mid \geq$

$\langle$binary arithmetic operator $\rangle$ ::= $+ \mid - \mid$ ***** $\mid$ **div** $\mid$ **mod**

$\langle$unary arithmetic operator $\rangle$ ::= $+ \mid -$

$\langle$predefined function call $\rangle$ ::=
 $\langle$predefined procedure name $\rangle$
 ($\langle$expression $\rangle$**,...,** $\langle$expression $\rangle$**)**

$\langle$variable name $\rangle$ ::= $\langle$identifier $\rangle$

All variable names and constants occurring in an arithmetic expression must be of integer type.

Let $\mathcal{E}$ be an expression evaluation context, we define the value of an arithmetic expression *wrt* $\mathcal{E}$ as follows. An integer constant is evaluated to itself. The value of a variable named *name* is the value of the variable designated by *name*. A parenthesized expression (*expr*) has the same value as the expression *expr*.

Let

- $expr_1$, $expr_2$ be two arithmetic expressions;
- ω an arithmetic operator.

The expression

$$expr_1 \ \omega \ expr_2$$

is evaluated as follows:

1. $expr_1$ and $expr_2$ are evaluated *wrt* $\mathcal{E}$. Let v_1 and v_2 be their respective values.
2. $v_1 \ \omega \ v_2$ is the value of the above expression.

Let *expr* be an arithmetic expression. The value of $-expr$ (resp. $+expr$) is $(-v)$ (resp. v) where v is the value of the expression *expr* *wrt* $\mathcal{E}$.

5.5.3 Relational expressions

Syntax

⟨relational expression ⟩ ::=
⟨arithmetic expression ⟩ ⟨relational operator ⟩ ⟨arithmetic expression ⟩

Let $\mathcal{E}$ be an expression evaluation context, we define the value of a relational expression *wrt* $\mathcal{E}$ as follows.

Let

- $expr_1$, $expr_2$ be two arithmetic expressions;
- ω a relational operator.

The expression

$$expr_1\ \omega\ expr_2$$

is evaluated as follows:

1. $expr_1$ and $expr_2$ are evaluated *wrt* $\mathcal{E}$. Let v_1 and v_2 be their respective values.
2. $v_1\ \omega\ v_2$ is the value of the above expression.

5.5.4 Conditions

Syntax

⟨condition ⟩ ::= **true**
| **false**
| **present** ⟨field name ⟩
| ⟨relational expression ⟩
| ⟨condition ⟩ ⟨logical operator ⟩ ⟨condition ⟩
| **not** ⟨condition ⟩
| **(** ⟨condition ⟩**)**

A name following the keyword **present** must designate a variable in $\mathcal{N}$(B1) *wrt* $\mathcal{E}$.

Semantics

Let $\mathcal{E}$ be an expression evaluation context, **true** and **false** evaluate to themselves. A parenthesized condition (*cond*) has the same truth value as *cond*. Let $cond_1$ and $cond_2$ be two conditions and ω a logical operator. The condition $cond_1$ **or** $cond_2$ evaluates to **true** if $cond_1$ or $cond_2$ evaluates to **true** *wrt* $\mathcal{E}$; it evaluates to **false** otherwise. The condition $cond_1$ **and** $cond_2$ evaluates to **true** if both conditions $cond_1$ and $cond_2$ evaluate to **true** *wrt* $\mathcal{E}$; it evaluates to **false** otherwise. Let *cond* be a condition. The value of **not** *cond* is **true** (resp. **false**) if *cond* evaluates to **false** (resp. **true**) *wrt* $\mathcal{E}$. Let *fname* designate a field name *wrt* $\mathcal{E}$, the condition

present *fname*

is evaluated to **true** if *fname* is present in the current record; it is evaluated to **false** otherwise. The expression *fname* has the value of the field *fname* in the current record if *fname* is present in the current record. The expression *fname* has an undefined value otherwise, but this causes no error.

Concrete Syntax

The previous description refers to the (simplified) abstract syntax of expressions. The actual concrete syntax is described below. It involves priority rules for the operators, allowing for a more convenient writing of the expressions.

⟨condition ⟩ ::= ⟨disjunction ⟩

⟨disjunction ⟩ ::= ⟨conjunction ⟩
| ⟨disjunction ⟩ **or** ⟨conjunction ⟩

⟨conjunction ⟩ ::= ⟨simple condition ⟩
| ⟨conjunction ⟩ **and** ⟨simple condition ⟩

⟨simple condition ⟩ ::= **true** |**false** | **present** ⟨field name ⟩
| ⟨relational expression ⟩ |(⟨condition ⟩)
| **not** ⟨condition ⟩

⟨relational expression ⟩ ::=
⟨arithmetic expression ⟩ ⟨relational operator ⟩
⟨arithmetic expression ⟩

⟨relational operator ⟩ ::= $< \,|> \,|\neq \,|= \,|\leq \,|\geq$

⟨arithmetic expression ⟩ ::=
⟨term ⟩
| ⟨arithmetic expression ⟩ ⟨additive operator ⟩ ⟨term ⟩

⟨term ⟩ ::= ⟨factor ⟩
| ⟨term ⟩ ⟨multiplicative operator ⟩ ⟨factor ⟩

⟨factor ⟩ ::= ⟨simple expression ⟩

⟨simple expression ⟩ ::= ⟨parameter name ⟩
| ⟨variable name ⟩
| ⟨integer_constant ⟩
| **(** ⟨arithmetic expression ⟩**)**
| ⟨pre-defined procedure call ⟩

⟨multiplicative operator ⟩ ::= * |**div** |**mod**

⟨additive operator ⟩ ::= + |-

It follows from this concrete syntax the operators priority defined hereafter:

priority	*operators*
1	$*$, **mod**, **div**
2	$+, -$
3	$<, >, \leq, \geq, =, !=$
4	**not**, **present**
5	**and**
6	**or**

5.6 Actions

The execution of an action is defined with respect to a given (initial) context and yields a new (final) context (unless it loops or yields an error).

5.6.1 Skip

skip is the empty statement, which is similar to the empty statement in Pascal. It can be useful in clarifying the code. Let $\mathcal{A}$ be an action execution context, the execution of the action **skip** *wrt* $\mathcal{A}$ does not modify $\mathcal{A}$.

5.6.2 Assignment

$\langle$assignment $\rangle$ ::= $\langle$ left expression $\rangle$ **:=** $\langle$right expression $\rangle$

$\langle$left expression $\rangle$ ::= $\langle$parameter name $\rangle$
  | $\langle$variable name $\rangle$

$\langle$right expression $\rangle$ ::= $\langle$expression $\rangle$

$\langle$expression $\rangle$ ::= $\langle$constant $\rangle$
  | $\langle$field name $\rangle$
  | $\langle$left expression $\rangle$
  | $\langle$expression $\rangle$ $\langle$binary arithmetic operator $\rangle$ $\langle$expression $\rangle$
  | $\langle$unary arithmetic operator $\rangle$ $\langle$expression $\rangle$

$\langle$field name $\rangle$ ::= $\langle$identifier $\rangle$

Let $\mathcal{A} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3)$ be an action execution context, and let *rexpr* be a right expression and *lexpr* a left expression according to the above syntax. The execution of the action

$$lexpr := rexpr$$

wrt $\mathcal{A}$ takes place according to the following steps:

1. Let *name* be the name of the variable designated by *lexpr* with respect to the expression evaluation context $\mathcal{E} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4)$. *name* must be a member of either sets B_2, B_3, or B_4; let B be such a set and $(name, v_0)$ the binding pair of *name* in B.

2. *rexpr* is evaluated *wrt* $\mathcal{E}$; let v_1 be its value (if this evaluation ever terminates). The value v_1 and the variable *name* must be of the same type.

3. As a result of executing the assignment *wrt* $\mathcal{A}$, the variable bindings B in $\mathcal{A}$ is replaced by the variable bindings $(B \setminus \{(name, v_0)\}) \cup \{(name, v_1)\}$.

5.6.3 Rule triggering

The BNF syntax of a rule triggering is as follows.

⟨rule triggering ⟩ ::= **trigger off** ⟨triggering mode ⟩ ⟨rule call ⟩

⟨triggering mode ⟩ ::= **for_next**
| **for_current**
| **at_completion**

⟨rule call ⟩ ::= ⟨rule name ⟩ **(** ⟨expression ⟩**,... ,** ⟨expression ⟩**)**

⟨rule name ⟩ ::= ⟨identifier ⟩

Let $\mathcal{A} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3)$ be an action execution context. The execution of the action

trigger_off *triggering_mode rule_name* ($expr_1$, ..., $expr_n$) (n≥0)

where

- *rule_name* is an identifier,
- $expr_1$, ..., $expr_n$ are expressions,
- *triggering_mode* ∈ {**for_current**, **for_next**, **at_completion**}

is as follows. Let R be the rule declaration designated by *rule_name wrt* $\mathcal{A}$. The arity of the rule *rule_name* must be n.

1. the expressions $expr_1$, ..., $expr_n$ are evaluated with respect to the expression evaluation context $\mathcal{E} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4)$ in an arbitrary order; let v_1, ..., v_n be the list of their respective values. The type of the values v_1, ..., v_n must be of the same types as the formal parameters appearing in R. Consider the rule instance

$$(\mathrm{R}, \mathrm{B} = \{(f_1, v_1), ..., (f_1, v_1)\})$$

where f_1, ..., f_n is the sequence of formal parameter names appearing in R. The execution of the rule triggering above is as follows:

2. (a) if *triggering_mode* = **for_current**
then the execution of the rule triggering above yields the action execution context $\mathcal{A}'$ defined by

$$\mathcal{A}' = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1 \cup \{(\mathrm{R}, \mathrm{B})\}, \mathrm{I}_2, \mathrm{I}_3)$$

In this case, we say that the rule *rule_name* is triggered for the current record.

(b) if *triggering_mode* = **for_next**
then the execution of the rule triggering above yields the action execution context $\mathcal{A}'$ defined by

$$\mathcal{A}' = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1, \mathrm{I}_2 \cup \{(\mathrm{R}, \mathrm{B})\}, \mathrm{I}_3)$$

In this case, we say that the rule *rule_name* is triggered for the next record.

(c) if *triggering_mode* = **at_completion**
then the execution of the rule triggering above yields the action execution $\mathcal{A}'$ defined by

$$\mathcal{A}' = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1\ , \mathrm{I}_2, \mathrm{I}_3 \cup \{(\mathrm{R}, \mathrm{B})\})$$

In this case, we say that the rule *rule_name* is triggered at completion time.

5.6.4 Predefined procedure call

The semantics of executing a predefined procedure or function call is target system-dependent and will be described in Section 6.5.

5.6.5 Conditional Actions

The syntax of a conditional action is as follows.

$\langle$conditional action $\rangle$ ::= **if** $\langle$guarded action $\rangle$ **;** **...** **;** $\langle$guarded action $\rangle$ **fi**

$\langle$guarded action $\rangle$ ::= $\langle$condition $\rangle$ $\rightarrow$ $\langle$action $\rangle$

Let

- $\mathcal{A} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4, \mathcal{R}_1, \mathcal{R}_2, \mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3)$ be an action execution context,
- $cond_1$, ..., $cond_n$ be a sequence of conditions,
- $action_1$, ..., $action_n$ be a sequence of actions (n$\geq$1),

the execution of the action

$$\begin{array}{l} \mathbf{if} \\ \quad \begin{array}{lcl} cond_1 & \rightarrow & action_1; \\ \vdots & & \vdots \\ cond_n & \rightarrow & action_n \end{array} \\ \mathbf{fi} \end{array}$$

wrt $\mathcal{A}$ is as follows:

1. $cond_1$ is evaluated with respect to the expression evaluation context $\mathcal{E} = (\mathrm{B}_1, \mathrm{B}_2, \mathrm{B}_3, \mathrm{B}_4)$; let v be its value.
2. if v = **true** $action_1$ is executed with respect to $\mathcal{A}$ and the execution is terminated; otherwise

   (a) if n > 1 the statement

   $$\begin{array}{l} \mathbf{if} \\ \quad \begin{array}{lcl} cond_2 & \rightarrow & action_2; \\ \vdots & & \vdots \\ cond_n & \rightarrow & action_n \end{array} \\ \mathbf{fi} \end{array}$$

   is executed with respect to the action execution context $\mathcal{A}$.

   (b) else (v = **false** and n = 1) the execution is terminated.

5.6.6 Repetitive Actions

The syntax of a repetitive action is as follows.

⟨repetitive action ⟩ ::= **do** ⟨guarded action ⟩ **;** **...** **;** ⟨guarded action ⟩ **od**

Under the conditions of the previous section, the execution of the statement

$$\begin{array}{l} \textbf{do} \\ \quad \begin{array}{lcl} cond_1 & \rightarrow & action_1; \\ \vdots & & \vdots \\ cond_n & \rightarrow & action_n \end{array} \\ \textbf{od} \end{array}$$

wrt $\mathcal{A}$ = (B_1, B_2, B_3, B_4, $\mathcal{R}_1$, $\mathcal{R}_2$, I_1, I_2, I_3) is as follows:

1. the conditions $cond_1$, …, $cond_n$ are successively evaluated with respect to the expression evaluation context $\mathcal{E}$ = (B_1, B_2, B_3, B_4) until one (if any) returns a true value. Let $cond_i$ ($1 \leq i \leq n$) be such a condition. In this case, $action_i$ is executed with respect to $\mathcal{A}$; let $\mathcal{A}_1$ be the resulting context. The whole statement

$$\begin{array}{l} \textbf{do} \\ \quad \begin{array}{lcl} cond_1 & \rightarrow & action_1; \\ \vdots & & \vdots \\ cond_n & \rightarrow & action_n \end{array} \\ \textbf{od} \end{array}$$

   is then executed over again with respect to $\mathcal{A}_1$.

2. if neither conditions $cond_1$, …, $cond_n$ is evaluated to **true**, the execution is terminated.

5.6.7 Compound actions

The syntax of a compound action is as follows.

⟨compound action ⟩ ::= **begin** ⟨action ⟩ **;** **...** **;** ⟨action ⟩ **end**

Let

- $\mathcal{A}$ be an action execution execution context,
- $action_1$, ..., $action_n$ a sequence of actions ($n \geq 0$).

The execution of the statement

begin $action_1$; ...; $action_n$ **end**

wrt $\mathcal{A}$ is terminated if $n = 0$ (i.e., the statement is **begin end**); otherwise, it is executed as follows:

1. $action_1$ is executed *wrt* $\mathcal{A}$; let $\mathcal{A}'$ be the resulting context,
2. the statement **begin** $action_2$; ...; $action_n$ **end** is recursively executed *wrt* $\mathcal{A}'$.

5.7 Processing a Record

Let $\mathcal{C}$ = (G, $\mathcal{R}$, I_1, I_2, I_3) be a record processing context and let CR be a record. The processing of CR is defined *wrt* the (initial) context $\mathcal{C}$ and yields a new (final) context $\mathcal{C}'$. We define the operational semantics of processing the record CR *wrt* $\mathcal{C}$ recursively on the size of I_1 as follows. If $I_1 = \emptyset$ the processing of CR *wrt* $\mathcal{C}$ is terminated. Otherwise the record CR is processed as follows.

1. Let (R, B) be an arbitrary rule instance in I_1. The rule body of R is executed with respect to the action execution context

$$\mathcal{A} = (\mathrm{CR}, \mathrm{G}, \mathrm{B}, \mathrm{L}, \mathrm{I}_1 \setminus \{(\mathrm{R}, \mathrm{B})\}, \mathrm{I}_2, \mathrm{I}_3)$$

   where L is the variable assignment over the set of local variables of R defined by:

$$\{(\mathrm{name}_1, \mathbf{undef}), ..., (\mathrm{name}_k, \mathbf{undef})\}$$

2. Let $\mathcal{A}'$ = (CR, G$'$, B, L$'$, I'_1, I'_2, I'_3) be the context resulting from executing R's body *wrt* $\mathcal{A}$. The record CR is then recursively processed *wrt* the context $\mathcal{C}'$ = (G$'$, $\mathcal{R}$, I'_1, I'_2, I'_3).

In other words, after executing a given rule instance (R, B) in I_1, it is removed from I_1 and we add to each of the multi-sets I_1, I_2, and I_3 whatever rule instances were triggered off by R for the current record, the next record, or at completion time, respectively.

Note

The order in which the rule instances in the set *Current* (denoted by I_1 above) are executed is undefined in the semantics since the process of selecting the rule instance to be executed next is not deterministically specified. Thus, the programmer should not assume any particular order of executing the current rule against the audit record. In particular, consider the case where we have n rule instances r_1, ..., r_n that modify a global variable say x, and an other rule instance r which then uses the value of x. The order in which r_1, ..., r_n are executed generally determines the final value of x. Let us assume that r_1, ..., r_n change the value of x according to some functions F_1, ..., F_n, respectively. To ensure that the order of executing r_1, ..., r_n does not affect the final value of x it is sufficient to ensure that:

$$F_{k_1} \circ ... \circ F_{k_n}(x) = F_{l_1} \circ ... \circ F_{l_n}(x)$$

where (k_1, ..., k_n) and (l_1, ..., l_n) are arbitrary permutations of $\{1, ..., n\}$ and $\circ$ denotes the function composition operator (f$\circ$g(x) = f(g(x))).

5.8 Processing a Sequence of Records

The processing of a sequence of records $\mathrm{CR}_1, \cdots, CR_n$ ($n \geq 0$) with respect to a given (initial) context $\mathcal{C}$ = (G, $\mathcal{R}$, I_1, I_2, I_3) yields a new (final) context $\mathcal{C}'$ and is defined recursively as follows. If $n = 0$, the processing is terminated. Otherwise, assume that we already processed a sequence of records $\mathrm{CR}_1, \cdots, CR_k$ ($1 \leq k < n$) with respect to $\mathcal{C}$ and let $\mathcal{C}'$ = (G$'$, $\mathcal{R}$, I'_1, I'_2, I'_3) be the context resulting from processing the record CR_k. We process the record CR_{k+1} with respect to the context $\mathcal{C}''$ = (G$'$, $\mathcal{R}$, I'_2, $\emptyset$, I'_3).

5.9 RUSSEL Programs

The logical structure of an audit file is defined as a sequence of records:

$$CR_1,\ ...,\ CR_n \qquad (\mathrm{n} \geq 0)$$

where all records in the sequence are bindings over a given fixed set of fields.

Consider a RUSSEL program composed of a sequence of modules M_1, ..., M_n. Let $\mathcal{R}_i$ be the set of internal rules of M_i and let Int_i be the set of internal global variable names declared in M_i. Let $\mathcal{R}$ be the set union of all external rule declarations contained in M_1, ..., M_n and let *Ext* be the set union of all external global variable names declared in M_1, ..., M_n. Since an init action declaration is optional, only a (possibly empty) subsequence M_{k_1}, ..., M_{k_m} $(m \leq n)$ of the sequence of modules M_1, ..., M_n contain an init action declaration. Let $init_{k_1}$, ..., $init_{k_m}$ be the corresponding init action declarations.

Having defined the RUSSEL program, we describe the operational semantics of executing the given RUSSEL program on an audit file as follows.

1. The variable bindings over the set *Ext* is initialized such that all the binding pairs have the value **undef**:

$$\mathrm{G} = \{(\mathrm{extvar}_1,\ \mathbf{undef}),\ ...,\ (\mathrm{extvar}_l,\ \mathbf{undef})\}$$

2. The variable bindings over every set Int_i is initialized such that all the binding pairs have the value **undef**:

$$\mathrm{G}_i = \{(\mathrm{intvar}^i_1,\ \mathbf{undef}),\ ...,\ (\mathrm{intvar}^i_{j_i},\ \mathbf{undef})\}$$

3. The set of rule instances I_1 is initialized to

$$\mathrm{I}_1 = \{(init_{k_1},\ \emptyset),\ ...,\ (init_{k_m},\ \emptyset)\}$$

4. We define a sequence of audit records

$$CR_0 = \{\},\ CR_1,\ ...,\ CR_m,\ CR_{m+1} = \{\}$$

5. We process the sequence of records CR_0, CR_1, ..., CR_m *wrt* the context $\mathcal{C}_0$ = (G, $\mathcal{R}$, I_1, $\emptyset$, $\emptyset$); let $\mathcal{C}'$ = (G$'$, $\mathcal{R}$, I'_1, I'_2, I'_3) be the resulting context. We then process the record CR_{m+1} *wrt* the context $\mathcal{C}''$ = (G$'$, $\mathcal{R}$, I'_3, $\emptyset$, $\emptyset$).

Chapter 6

Unix Level Description of ASAX

6.1 Introduction

The previous description of RUSSEL rules out a number of details which are needed to use the language in the context of a particular operating system-specific environment. We now provide the additional pieces of information that are needed to run (our implementation of) the language under the UNIX operating system. In particular, we describe precisely the physical representation of NADF files, the mapping between external (human readable) field names and their internal representation. We additionally provide the exact description of the interface with the C-language supporting predefined routine calls. We depict how argument values are passed from RUSSEL to an external predefined routine and the steps involved in incorporating user-defined routines into RUSSEL. To ensure the portability of ASAX, we hide as much as possible the implementation details of NADF records input to the RUSSEL engine. These implementation details are hidden into well specified input routines, called *virtual routines* for audit record input.

This chapter is organized as follows. Section 6.2 contains the physical layout of NADF records and NADF files. Section 6.3 presents how the external to internal mapping of field names is represented. Section 6.4 describe the virtual input routines. Section 6.5 describes the interface with C. In Section 6.6, we present the user command line interface. Finally, Section 6.7 overviews the main implementation issues.

6.2 Physical Representation of NADF files

In the previous chapters, we only provided the logical structure of audit files. Recall from the previous chapter that an audit file is a sequence of records, where all the records are variable bindings over some subset of field names and each field (also referred to as *audit data field*) is a pair (*fname*, *value*), where *fname* is the name of the field and *value* its value.

In the present section, we describe the exact NADF physical representation of an audit file. We proceed by describing the representation of a field, a record, and finally the whole NADF file. However, we first associate with each field name a (small) natural number, called the *audit data identifier*, that unambiguously identifies each field name among all other field names. Note that audit data identifiers are assigned arbitrarily, provided they are all distinct. Moreover each NADF record may contain a different number of audit data fields. Record fields are referenced in RUSSEL modules by their field names and are represented internally by the corresponding audit data identifiers.

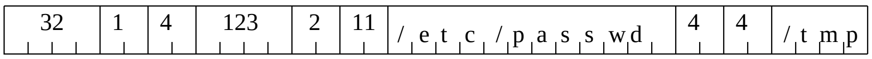

Figure 6.1: NADF record

Representation of Audit Data
Let (*fname*, *value*) be a field, its representation is an area consisting of the following contiguous items:

identifier the unsigned short (2 bytes) binary representation of the audit data identifier associated with the field name *fname*,

length the unsigned short (2 bytes) binary representation of the length in bytes of the field's value *value*,

value the byte string representing the value *value* itself.

Audit Data Alignment
Audit data values with an odd number of bytes are appended with a space character (32 in ASCII code) to ensure that the representation of the next audit data is 2-bytes aligned. Furthermore, the padding byte is not taken into account in calculating the length of the audit data value.

Representation of a record
The NADF representation of a record is an area which consists of the following contiguous items:

total record length A four bytes binary representation of the length in bytes of the whole NADF record (including the record length),

sequence of fields The contiguous sequence of representation of each field in the record; the sequence is sorted in ascending order of audit data identifiers.

Example
Suppose we want to convert to NADF format a record having the C declaration:

```
struct {
     char directory[10];
     int uid;
     char filename[12];
} Record = {"/tmp", 123, "/etc/passwd"};
```

and suppose the audit data fields *directory*, *uid*, and *filename* are assigned the identifiers 4, 1, and 2 respectively. The corresponding NADF record is depicted in Figure 6.1. Note in particular that strings in an NADF record are *not* null-terminated since the audit data length is used instead.

Finally, the physical representation of an NADF file is a sequential file composed of the sequence of NADF records. Moreover, a NADF file must always begin with a header record having the following C-declaration:

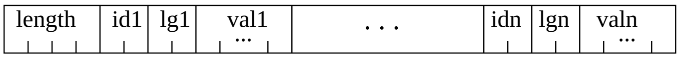

Figure 6.2: General NADF record layout

```
struct {
     int len;
     char val[12];
} TypeNADF = {15,"__NADF__1| \0"};
```

Before NADF files are processed by ASAX, the existence of the header record is always checked in order to avoid processing non-NADF files.

NADF record alignment

Since most modern machine architectures require 4-bytes integers to be 4-bytes aligned, a NADF record must always begin at a 4-bytes aligned area. This is especially important if NADF records are buffered for I/O. Similarly, if the total number of bytes in a NADF record is not a multiple of 4, padding bytes (spaces) must be added at the end of the record to ensure that the next record is aligned on 4-bytes boundaries. Again, the padding bytes are not taken into account in calculating the record total length. Figure 6.2 depicts the general layout in memory of a NADF record.

6.3 Audit Data Description Files

In RUSSEL, audit data fields are represented externally as human readable names (e.g., SUBJ_UID, SUBJ_PID, HEADER_TIME) and are represented internally as audit data identifiers in NADF files (e.g., the identifiers of SUBJ_UID, SUBJ_PID, and HEADER_TIME could be 1, 23, and 123 respectively). External field names are convenient for writing RUSSEL modules whereas internal audit data identifiers are useful to improve internal processing of audit records. Consequently, the table defining the mapping between external and internal representation of audit data is stored in a particular file called *audit data description file* In practice, this also means that the task of a format adaptor is not restricted to the conversion of audit trails to NADF format. Thus, a format adaptor for a particular kind of sequential file (BSM audit trails, BS2000 audit trails, IP network packet stream, etc.) also aims at generating an audit data description file to store such a mapping. During the processing of RUSSEL modules, ASAX needs the mapping between external and internal audit data representation. An audit data description file is a text file which mainly specifies the mapping between audit data field names and audit data identifiers. In an is described by a sequence of five lines[1]. For easier parsing, the first line begins with '1' followed by a blank, the second line begins with '2' followed by a blank and so on. The first line in each sequence indicates the audit data identifier, the second indicates the type of the audit data in the native record. The third one indicates the type of the audit data in the NADF record. The second and third line are not interpreted for the present ASAX version but are useful in order to keep track of the original data type in the native audit records and the corresponding data types in the NADF file. The fourth line indicates the data field name and finally, the fifth line contains a free comment about the meaning of the audit data and is not interpreted by ASAX. A sequence of header lines (depicting

[1]Certain lines are provided for historical reasons and are irrelevant for our purpose.

for instance the version number, machine type, operating system type, etc.) can optionally appear at the beginning of the file. These lines must appear as follows: 0 or more lines beginning with letter 'A' followed by 0 or more lines beginning with letter 'B', and so on up to 'F'. A sample data description file is shown in Section A.3.

6.4 Specification of the Virtual Interface

The purpose of the virtual input routines is to hide from RUSSEL's engine all the details related to reading NADF audit records from a specific source. For instance, NADF records can originate from the local file system, a remote machine, or from merging several input streams originating from several hosts in a distributed environment. Moreover, audit record input may be done on-line or off-line depending on the security administrator's needs. Additionally, audit records may originate from other audit trail analysis systems in an inter-operable environment. Since it is not possible to accommodate all these specific sources, we define three routines (*vopen*(), *vread*(), and *vclose*()), for ensuring audit data source independence of ASAX. It is up to the user to implement these three routines; their specification is provided below. Using the command `masax` (see Section 6.6.2), the C object module implementing the virtual routines is then linked with ASAX to support the analysis of the corresponding audit data source.

Informally, a virtual open (in read mode) performs whatever actions are needed to prepare a virtual file for reading. A virtual file may be the result of merging several audit files generated by different severs on the network.

The virtual open routine has the prototype `int vopen (int argc, char ** argv)`. It receives a list of file names and builds a single file resulting from an unspecified set of operations over the named files. The resulted file is open in a virtual read mode. `argv` is a pointer to an array of character strings that contains `argc` arguments, one per string. After a successful virtual open, a stream of records is opened for reading i.e., the next call to *vread*() is able to read the first record in the stream if there is one.

A virtual read simply returns the address of the next NADF record. The prototype of the virtual read routine is `int vread (char ** ptr)`. Its purpose is to return the address of the next NADF record in the stream. A stream must be previously opened as in the postcondition of a *vopen*() call. On success, `*ptr` is the address of the next NADF record in the stream.

A virtual close has the prototype `int vclose()`. Its purpose is to close the currently open virtual audit file previously open by *vopen*(). All three routines return 0 on success and -1 on failure.

It should be stressed that the specifications of the virtual routines are just requirements that an implementor must meet, not functionalities provided by ASAX.

6.5 Interface with C

An important goal which has been pursued in designing the language RUSSEL is to provide the user with the ability to extend the capabilities of the language through the incorporation of user-defined routines. Indeed, writing RUSSEL modules may require a functionality which is not directly implementable in RUSSEL. Example such functionalities include converting a user Id to the corresponding user name, checking file attributes, converting an integer representation of time to a human readable formatted string, and simulating complex data structures. Obviously, no matter how comprehensive the available predefined routines are, there always could be a situation where a predefined routine is needed which is not supported in RUSSEL.

The purpose of this section is to describe how RUSSEL is interfaced to a general purpose programming language, namely C. Section 6.5.1 contains a description of the step involved in adding a user-defined routine to the language. Section 6.5.3 describes how the argument values are layed out in a shared memory area which is passed from RUSSEL to a user-defined routine and vice-versa.

6.5.1 Preparing a C-module linkable with ASAX

In order to add a new user-defined routine to the language, the user is required to follow the two steps explained hereunder.

1. Using the command `libman` described in Section 6.6.1, the user supplies a high-level description of the C-routines s/he wishes to add to the language. For each added routine, this description contains the routine name, the type and passing modes of each parameter of the C-routine, the type of returned value, and finally, the name of the C-object module implementing the C-routine.

2. Using the command `masax` described in Section 6.6.2, the user links ASAX's object module with the object module of the C-routine in order to generate a new ASAX executable program that supports predefined routine calls to the added C-routine.

In performing the two steps above, the user should not be required to know about ASAX's implementation details such as the internal data structures and code that are related to the internal representation and management of the C-routines; these implementation details are hidden as much as possible. Moreover, the second step above is automated and only the high-level description of the C-routines are needed in order to generated a new ASAX executable program. Instead of directly modifying ASAX source files, when a user is to add a new C-routine to the existing library, s/he is provided with a small and simple language that allows him/her to describe these C-routines and let an ASAX's library manager program rebuild a new asax executable on behalf of the user. The set of predefined routines is henceforth referred to as the *C-library*. The two steps outlined above are detailed in the next two sections.

6.5.2 Library Description Files

The high-level description of the C-library aims at providing ASAX with all pieces of information that are needed for invoking a C-routine in a RUSSEL module. These pieces of information are: the type of the returned value which is either string or integer, the name by which the routine is called in RUSSEL, the type and passing mode of each parameter, and finally the C-object module that implements the C-routine. The passing modes are similar to Pascal's parameter passing modes and may either be by value or by reference. It is also possible to describe a C-routine which has a variable number of arguments, in which case all parameters are required to have the same passing mode.

A *library description file* is a text file which contains for each C-routine in the library a line that specifies the pieces of information comprising its high-level description. We provide a simple syntax for specifying the high-level description of the C-library which is given below.

```
int closeNADF(int:val):lib;
int creatNADF(str:val):lib;
display_current():show;
print(undef):lib;
println(undef):lib;
int strToInt(str:val):lib;
updstat(str:val, str:val, str:val, str:val):stat;
int writeNADF(int:val):lib.
```

Figure 6.3: library description file

$\langle$command file$\rangle$ ::= $\langle$empty$\rangle$
| $\langle$descriptor list$\rangle$

$\langle$descriptor list$\rangle$ ::= $\langle$descriptor$\rangle$ **;** **...** **;** $\langle$descriptor$\rangle$ **.**

$\langle$descriptor$\rangle$ ::= $\langle$returned type$\rangle$ $\langle$func_proc_name$\rangle$
$\langle$parameter type list$\rangle$ $\langle$filename$\rangle$

$\langle$returned type$\rangle$ ::= $\langle$empty$\rangle$
| **int**
| **str**

$\langle$func_proc_name$\rangle$ ::= identifier

$\langle$parameter type list$\rangle$ ::= $\langle$empty$\rangle$
| $\langle$variable_arity$\rangle$
| $\langle$param_list$\rangle$

$\langle$variable_arity$\rangle$ ::= **undef**

$\langle$param_list$\rangle$ ::= $\langle$parameter type$\rangle$ **,** **...** **,** $\langle$parameter type$\rangle$

$\langle$parameter type$\rangle$ ::= $\langle$type$\rangle$ **:** $\langle$passing mode$\rangle$

$\langle$type$\rangle$ ::= **int** | **str**

$\langle$passing mode$\rangle$ ::= **val** | **ref**

$\langle$filename$\rangle$ ::= identifier

Figure 6.3 depicts an example library description file.

6.5.3 Conventions for Argument Passing

The purpose of this section is to give a precise specification of the interface between the evaluator and a C-routine. This is a crucial point since the user writing his/her own C-routine to be integrated with the existing library must supply the arguments in a well defined format in order to be handled properly by the evaluator. Conversely, an argument area supplied by the evaluator must be correctly interpreted by a C-routine.
Since a C-routine may take a variable number of arguments, rather than supplying a list of arguments, a pointer to a memory area containing the list of arguments is supplied instead. As a result of this choice, two points have to be specified:

- how ASAX data types (**integer** and **string**) are actually represented;
- how these data are organized in the arguments area so they can be parsed correctly by a C-routine.

ASAX data representation: The integer type in ASAX is simply represented as a 4-bytes **int** in C (the evaluator is written in C). As a result, since ASAX code is supposed to be completely portable, the user bears the responsibility to choose the appropriate scalar type (**short**, **int**, **long int**) having 4 bytes in size. This integer-size assumption is necessary in order to avoid problems when porting ASAX to an architecture with a different word size characteristics.
A string s of length l is represented by a pointer to a memory area of two fields. The first one is the length of s and the second is the string itself which is 2*((l+1) div 2) bytes long. This area is always aligned on 2 bytes boundaries.

Format of an argument area: A predefined C-procedure $p(p_1, \ldots, p_n)$ is implemented by a C-routine which takes a single argument which is the address of a memory area representing the argument list. This area is always aligned on a 4 bytes boundaries for compatibility reasons and is organized as follows:
Consider a given argument p_i. If p_i is a value argument:

- of type **integer**, p_i is this integer;
- of type **string**, p_i is the pointer representing this string.

if p_i is a reference argument:

- of type **integer**, p_i is a 4-bytes pointer to a 4-bytes **int**;
- of type **string**, p_i is a 4-bytes pointer to a 4-bytes aligned area.

The value returned by a C-routine implementing a predefined function is the one returned by this function and this is always ensured by the evaluator. The example of Figure 6.4 illustrates the four above cases.

Implementation of a predefined function or procedure Let $p(v_1, \ldots, v_n; r_1, \ldots, r_n)$ be a predefined function or procedure, where $v_1, \ldots, v_n$ are value parameters and $r_1, \ldots, r_n$ are reference parameters. p is implemented by a C-routine of the same name (see Figure 6.5), where s is the address of the argument area. This area is scanned from its address and since we know the arity of the implemented function or procedure and the type and the passing mode of each argument, it is possible to fetch any of these arguments starting from the address s.

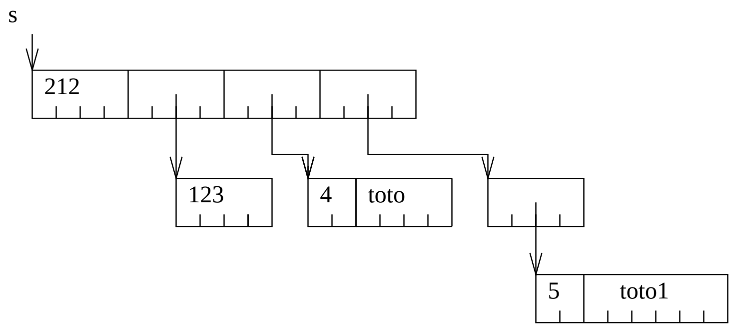

Figure 6.4: Example Argument Area

```
p(char *s) {
...
...
...
}
```

Figure 6.5: A schema for a user-defined routine

6.6 Command Line Languages

6.6.1 The C-Library Manager

SYNOPSIS

`libman [C-lib]`

DESCRIPTION

`libman` receives a high level description of the C-routines to be integrated with ASAX. Such a description is contained in an ASCII file. A C-routine can be added, modified or removed from an ASAX version simply by adding, modifying or removing its description from the library description file. `libman` adds the user-defined C-routines to the predefined C-routines. It assumes that the file with the path name `C-lib.ldf` contains the high level description of the C routines to be added. If no `C-lib` argument is supplied, the file `ldf` in the current directory is taken by default. `libman` generates three files (see FILES below), which collectively constitute the internal representation of the C-library. This internal representation is subsequently used by the command `masax` (see Section 6.6.2) which generates a new ASAX version.

FILES

`[C-lib.]asax_lib.h` contains the internal descriptors of the C-routines, `[C-lib.]proto.h` contains the prototypes of the C-routines, and `[C-lib.]makeAsax` is a script file to generate a new ASAX version.

DIAGNOSTICS

If the library description file does not respect the library description file BNF syntax, one or more error messages are displayed on the standard output.

6.6.2 ASAX version generator

SYNOPSIS

`masax [-sc] [-lib Clib] [-fa io-obj] [asax-executable-file]`

DESCRIPTION

The command `masax` aims at generating a new ASAX executable file. Two components can be modified in an ASAX version: the supported C-library routines and the virtual I/O routines on the native files to be analyzed. The C-library object files and I/O routines on analyzed files are linked with the ASAX kernel to obtain a new executable file. `masax` optionally (`-lib`) uses the internal representation of such C-library routines to find out the required object modules and how to integrate them in the new ASAX version. The new ASAX version can operate in three different modes: the *standard* (option `-s`) the conversion/evaluation (option `-c`) and the mixed modes (option `-sc`) depending on the format of the analyzed file.

OPTIONS

-s The ASAX version generated by this option assumes that analyzed files are in NADF format. This is the default option and corresponds to the standard mode.

-c The ASAX version generated by this option operates in the conversion/evaluation mode. In this mode, records are read from the analyzed file and converted to NADF format before being submitted to the evaluator for analysis. With this option, the user is supposed to supply three virtual routines on the native file. The object module for these routines is specified using option `-fa`.

-sc The ASAX version generated by this option operates in the mixed mode which supports both the standard and conversion/evaluation modes. With this option also, the user is supposed to supply three I/O routines on the native file. The object module for these routines is specified using option `-fa`.

-lib Clib Generates an ASAX version that supports the C-library whose internal representation is identified by the prefix `Clib`.

-fa io-obj Generates an ASAX version where virtual I/O routines (vopen(), vread() and vclose()) are in the object file `io-obj.o`

The generated executable file is `asax-executable-file` if such argument is given and is `asax.exec` by default.

EXAMPLES

To generate an ASAX version `asax.std` in standard mode supporting the C-user-defined library identified by the prefix `usr-lib` (previously generated using `libman`:

```
example% masax -s -lib usr-lib  asax.std
ASAX linkage is under way. Please wait ...
example%
```

To generate a version `asax.conv` in conversion/evaluation mode with only the predefined routines and where the object module `vio.o` implements I/O routines on the analyzed file:

```
example% masax -c -fa vio asax.conv
ASAX linkage is under way. Please wait ...
example%
```

To generate a version asax.mix in mixed mode where the C-user-defined library is identified by the prefix `usr-lib` and the I/O routines object module is `vio.o`:

```
example% masax -sc -lib usr-lib -fa vio asax.mix
example%
```

DIAGNOSTICS

If the library description file does not respect the library description file BNF syntax, one or more error messages are displayed on the standard output.

6.6.3 ASAX Command Line

SYNOPSIS

`asax-executable-file [-sclg] addf russel-program [audit-file]`

DESCRIPTION

`asax-executable-file` is the ASAX executable file, `addf` is the audit data description file (see Section 6.3), `russel-program` is the RUSSEL program which must have the extension `.asa`, and `audit-file` is the file to be analyzed with ASAX. The ASAX executable `asax-executable-file` can operate in three different modes: the *standard* (option `-s`) the conversion/evaluation (option `-c`) and the mixed modes (option `-sc`) depending on the format of the analyzed file. Option `-s` (the default one), invokes ASAX in standard mode. Option `-c` is only relevant if `asax-executable-file` was previously generated for the mixed mode using `masax` with option `-sc` (see Section 6.6.2). If `audit-file` is in NADF format, `asax-executable-file` is assumed to be in standard mode or mixed mode. Otherwise, if `audit-file` is not in NADF format, `asax-executable-file` is assumed to be in conversion/evaluation mode or mixed mode. In the latter case, option `-c` must be supplied.

OPTIONS

-l Performs only a syntactic check of the RUSSEL program `russel-program`; all modules possibly used by `russel-program` are ignored. With this option, the audit file argument is ignored.

-g Performs only a syntactic check of the RUSSEL program `russel-program` as well as all modules possibly used by `russel-program`. With this option, the audit file argument is ignored.

-c `asax-executable-file` is assumed to be in mixed mode. The RUSSEL program is syntactically checked and if no error is found, the audit file is analyzed in conversion/evaluation mode.

-s This is the default option. `asax-executable-file` is assumed to be in standard or mixed mode. The RUSSEL program is syntactically checked and if no error is found, the audit file which is assumed to be in NADF format is analyzed in standard mode.

DIAGNOSTICS

If syntactic errors are found in either the audit data description file `addf` or the RUSSEL program `russel-program`, ASAX prints error messages describing these errors on the standard output.

6.7 Implementation Issues

6.7.1 Introduction

The implementation issues of the language are carefully addressed bearing in mind that efficiency is a key and a critical requirement for ASAX to be useful in a real life operating environment. Indeed, failure to meet the efficiency requirement simply makes the expressive power of the rule-based language RUSSEL an unusable feature. By examining the very nature of processing complex record patterns in an audit trail, there are two major issues that constitute a potential source of performance degradation. The first issue is the evaluation of conditions because most of module execution time is spent in verifying the eligibility of a detection rule to be triggered based on evaluating guard expressions. The second issue which is closely related to the first one is the fast access to record fields, because most guard expressions are formulated in terms of the current record fields.

The rest of this section is organized after the two issues outlined above. Section 6.7.2 provides a detailed architecture by describing how rules are executed by the evaluator and which are the data structures involved. Section 6.7.3 describes the internal code for representing conditions and guarded actions. Finally, Section 6.7.4 overviews how the current record is efficiently accessed by the abstract machine interpreter.

6.7.2 Internal Architecture

Each rule declared in a RUSSEL module is compiled to an abstract machine code which is then executed by the evaluator. Four storage areas are statically allocated at compile-time for storing external and internal global variables, actual parameters and local variables (see Figure 6.6). At the time the modules are compiled and an internal code is generated for them, every reference to a variable is represented as an address within the allocated storage area corresponding to the variable. As a result, the absolute addresses of variables are completely determined at compile-time, and the value of a variable is readily available at the cost of a single pointer dereferencing. An additional statically allocated area called the *working area* is utilized by the evaluator for storing intermediary values while evaluating arithmetic expressions.

The three sets of rule instances, denoted by I_1, I_2, and I_3 in the operational semantics of RUSSEL (see Section 5.7) are represented by three linked lists (pointed by `Current`, `Next`, and `Compl` in the figure). Every time a rule is triggered, the resulting rule instance is stored in a freshly allocated memory area containing two pointers: *i/* a pointer to the rule's abstract machine internal code, *ii/* a pointer to a dynamically allocated area holding the actual parameter values that are passed to the rule instance. Notice that several instances of the same rule may exist (with perhaps different parameter values) and all of these instances share the same internal code.

The processing of the current record (see Section 5.7) is performed by executing successively the internal codes of every rule instance in the list `Current`. Prior to executing the internal code of a rule instance, the actual parameters are copied into the static area `Par`. This copying operation is the price to pay to have direct access to the actual parameters when executing the internal code of every rule.

Upon termination of the current record processing, i.e., when the list `Current` becomes empty, the next record is read and the pointers `Current` and `Next` are simply swapped so that the rule instances that were triggered for the next record at the time the previous record was processed are now executed.

Finally, upon encountering the end of the audit file, the rule instances in the list `Compl` are executed to conclude the analysis.

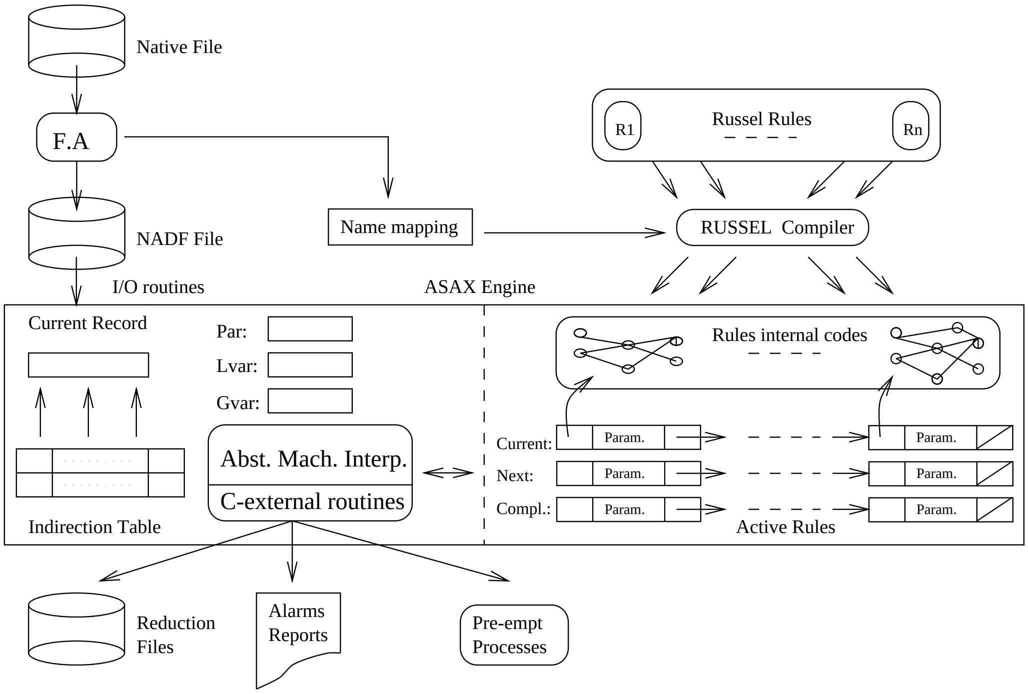

Figure 6.6: Architecture of the Evaluator

6.7.3 Internal Code

We only focus here on the efficiency of the abstract machine internal codes which are generated for evaluating conditions. Moreover, we present an example RUSSEL program fragment and the corresponding abstract machine internal code in order to illustrate the technique involved.

The main implementation (although classical) idea underlying efficient condition evaluation consists of generating binary tree internal code where each node in the tree is responsible for evaluating a simple condition (a relational expression or a test for the presence of a field name using the construct **present** *fname*). This way, most elementary conditions are not evaluated at all in many cases, thereby providing early detection of the truth value. This implementation technique of condition evaluation can be very rewarding.

Besides the relational operator involved and the location of the operands, the cell representing a simple condition, say *cond*, has two branch addresses: one branch is the address of the next simple condition to be evaluated if *cond* is evaluated to true and the second branch is the address of the next simple condition to be evaluated if *cond* is evaluated to false. The way branch addresses are connected depends on which logical connectives (and/or) are used to build the compound condition.

Example internal code

The rule depicted in Figure 6.7 is very simplified and this is only to illustrate abstract machine internal codes. The rule prints an alarm message whenever a user logs on to the system after a time limit denoted by *timelmt* or if the (suspected) user *joe* logs on at any time. In either case, the rule sets on the global variable denoted by *flag*.

```
global flag: string;
rule joe(timelmt: integer);
begin
   if
    EVENT = 'login' and
    present TIMESTAMP and
    TIMESTAMP > timelmt
                  → print('Late Night login by ', USERNAME);
    USERNAME = 'joe'
                  → print('Joe''s login attempt detected')
   fi;
   flag := 'on'
end
```

Figure 6.7: Example Rule

The corresponding internal code is given in Figure 6.8. Note in particular that if the condition `EVENT = 'login'` fails, none of the remaining conditions are evaluated and in this case, the condition `USERNAME = 'joe'` is evaluated next.

6.7.4 Access to the Current Record

The second implementation issue aims at ensuring a fast access to the current record fields and is explained in the following.

According to the description of the physical representation of NADF records presented in Section 6.2, a record is composed of a sequence of a variable number of audit data fields. Bearing in mind this fact, the objective of the implementation is to ensure that every data field is accessed in a constant time in spite of its variable location in the record.

At compile-time, an indirection table is allocated statically where an entry is composed of two elements. The first element is an audit data identifier and the second element is a pointer that will ultimately be assigned the address of the corresponding audit data value within the current record buffer. During the parsing of a RUSSEL module, the audit data identifiers corresponding to only those field names referenced in the rules of the program are stored in the indirection table

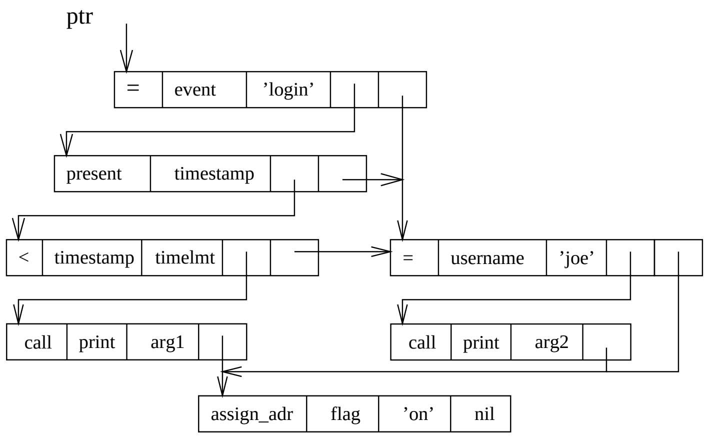

Figure 6.8: Example internal code

in ascending order. Additionally, references to audit data values in the internal code are translated into the address of the second element of the corresponding indirection table entry.

Every time a new record of the audit trail is read, it is scanned in parallel with the indirection table in order to update references to the data values. Given an audit data identifier in the indirection table, if the current record contains the corresponding audit data, then a reference to the audit data is added into the indirection table. Otherwise, a reference to a special area containing an empty string is put into the indirection table. Notice that it is important to distinguish between an audit data field which is present in the current record yet has zero bytes and an audit data field which is not present in the current record and the value of which is the zero byte string located in the static area. This allows the evaluator to resolve references to audit data fields exactly in the same way, whether the referenced field is present in the current record or not and a reference to a non present audit data field causes no runtime error.

Chapter 7

Finite State Automata

7.1 Introduction

The motivations for this chapter are the following. Many intrusion patterns can be described by means of regular expressions or finite state automata. Thus, the translation scheme presented in this chapter provides a basic methodology for building RUSSEL programs from higher level descriptions of intrusion patterns. However, in general, intrusion patterns cannot be completely described by regular expressions or finite state automata since the records in the sequence to be found are related by various constraints such as containing a common value, being included within a given time window, etc. These constraints can be taken into account by adding parameters and tests to the rules, but they do not require a major change to the methodology. Finally, since it is well known [39, 40, 31] that nondeterministic finite state automata are more convenient than deterministic ones to solve matching problems, it was interesting to show that such automata are straightforwardly translated into RUSSEL.

Section 7.2 introduces regular expressions and finite state automata. Section 7.3 contains the translation rules for converting an automaton to a RUSSEL module. In Section 7.4, we extend our translation rules to the case where the automaton is a nondeterministic one. Finally, Section 7.5 contains the conclusion.

7.2 Regular Expressions and Finite State Automata

7.2.1 Regular Expressions

Let Σ denote a set of symbols, called *letters*, and Σ^* denote the set of all finite sequences of letters over Σ, called *words*. The abstract syntax of a regular expression is defined as follows

$$r \ ::= a \ \mid r\,r \ \mid r + r \ \mid r^*$$

where a denotes an arbitrary letter in Σ. The semantics of a regular expression over Σ is defined by a function f that maps a regular expression to a set of words in Σ^*, i.e., $f\colon RE \rightarrow \wp(\Sigma^*)$, where $\wp(\Sigma^*)$ denotes the set of all subsets of Σ^*, and RE denotes the set of regular expressions. We define f recursively on the syntactic structure of a regular expression as follows:

- $f(a) = \{a\}$
- $f(r_1\,r_2) = \{w_1\,w_2 \dagger w_1 \in f(r_1) \wedge w_2 \in f(r_2)\}$

- $f(r_1 + r_2) = f(r_1) \cup f(r_2)$
- $f(r^*) = \{w \dagger \exists n \in \mathcal{N} \; \exists w_1, ..., w_n \in f(r) : w = w_1 \cdots w_2 \; \}$

where a denotes an arbitrary letter and r, r_1, r_2 denote regular expressions. Given a regular expression, the set $f(r)$ is called the language of r.

Example Let $\Sigma = \{a, b\}$, we have $f(a+b) = \{a, b\}$ and $f(ab^*) = \{a, ab, abb, \cdots\}$.

7.2.2 Finite State Automata

A finite state automaton M is a 5-tuple $(S, s_0, A, \Sigma, \delta)$, where

- S is a finite set of *states*,
- $s_0 \in S$ is *the start state*,
- $A \subseteq S$ is a distinguished set of *accepting states*,
- Σ is a finite *input alphabet*,
- δ is a relation from $S \times \Sigma$ to S, which is defined everywhere in $S \times \Sigma$, i.e., for each pair (s, a) $\in S \times \Sigma$, there exists at least one state $t \in S$ such that (s, a) δ t. δ is called the *transition relation* of M.

A finite state automaton defines a state-transition graph where the nodes are the states of the automaton and there is a transition from s to t labeled by a if $(s, a) \, \delta \, t$.

Example The state-transition diagram shown in Figure 7.1 is defined by the finite state automaton $M = (\{s_0, s_1\}, s_0, \{s_1\}, \Sigma = \{a, b\}, \delta)$, where $\delta(s_0, a) = \delta(s_0, b) = \delta(s_1, a) = \delta(s_1, b) = s_1$. Conventionally, we indicate the start state by an arrow entering from nowhere whereas accepting states are represented by thick circles. This convention will be used throughout the rest of this chapter.

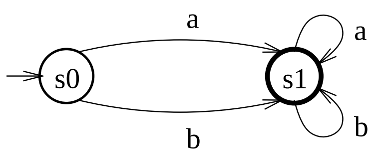

Figure 7.1: Example automaton

A finite automaton $M = (S, s_0, A, \Sigma, \delta)$ is said to be *deterministic* if δ is a (total) function, i.e., all transitions leading out from a given state are labeled by a distinct symbol. A *nondeterministic* automaton may contain a state s with two or more transitions leading out from s which are labeled by the same input symbol.

The execution of an automaton on a given input string proceeds as follows. The automaton starts in the start state s_0 and reads the letters of its input string one at a time. When a deterministic automaton is in a state s and reads a letter a, then the automaton makes a transition out of state s labeled by the letter a, to the state $\delta(s, a)$. Whenever a deterministic automaton reaches an accepting state, i.e., a state in A, we say that the automaton has *accepted* the input sequence read so far. An input sequence that is not accepted is said to be *rejected* by the automaton. Now we describe how nondeterministic automata are simulated. When a nondeterministic automaton receives a letter a that labels more than one transition out of a given state s, the automaton non-deterministically makes all the transitions labeled by a to the next states and will be in all output states of these transitions at the same time. Let D be the set of states that are reached by a nondeterministic automaton M after reading an input sequence w. M is said to have *accepted* the input sequence w if there is at least one state $s \in D$ such that s is an accepting state; we say that the automaton has *rejected* the input sequence w otherwise.

The example automaton shown in Figure 7.1, accepts all non empty input sequences of a and b. Given a word $w \in \Sigma^*$, the final state reached by a deterministic automaton after reading the letters in w one after another is denoted by $\phi(w)$, where the function ϕ is recursively defined by (ϵ denotes the empty word formed of no letters):

- $\phi(\epsilon) = s_0$,
- $\phi(wa) = \delta(\phi(w), a), \qquad for\ w \in \Sigma^*, a \in \Sigma$

ϕ is called the *final-state function*.

7.3 Simulating Finite State Automata with RUSSEL

In this section, we present translation rules that allow one to derive a RUSSEL program from a finite state automaton based on the corresponding state transition graph. We show that the derived RUSSEL program allows us to match the same pattern as the automaton. In order to simulate the behavior of the automaton using the RUSSEL program, the input sequence of the automaton is encoded in a NADF file containing the same sequence of symbols. Technically, the input sequence is represented by an "audit trail" whose records consist of a single field e. Thus, the i^{th} symbol of the input sequence is equal to the value of the field e in the i^{th} record of the "audit trail".

7.3.1 Translation rules

Let $M = (S, s_0, A, \Sigma, \delta)$ be an automaton. We derive a RUSSEL program P from M according to the following rules:

1. P contains an integer global variable say nar,
2. If the start state is not an accepting state (i.e., $s_0 \notin A$), then P's init action declaration is

```
init_action;
begin
   nar := 0;
   trigger off for_next s0;
   trigger off at_completion result
end
```

   Otherwise P contains the same init action declaration above 0 is replaced by 1. The rule *result* is declared as a follows.

```
rule result;
if nar > 0 --> print('YES'); true --> print('NO') fi;
```

3. P contains a rule declaration for each state in S. To a state $s \in S$ with successor states s_1, ..., s_n (i.e., for $i \in \{1, ..., n\}$ there exists $e_i \in \Sigma$ such that $\delta(s, e_i) = s_i$) corresponds a rule declaration in P of the same name as s which has no parameter and no local variable declarations and which has the following action part

```
begin
   a0;
   if e = 'e1' --> begin a1; trigger off for_next s1 end;
                      ⋮
     e = 'en' --> begin an; trigger off for_next sn end
   fi
end
```

the action a_0 is equal to `nar := nar - 1` if $s \in A$ and to **skip** otherwise. Each action a_i ($1 \leq i \leq n$) is equal to `nar := nar + 1` if $s_i \in A$ and to **skip** otherwise.

4. P contains no other rule declarations.

A rule which corresponds to an accepting state is said to be an *accepting* rule.

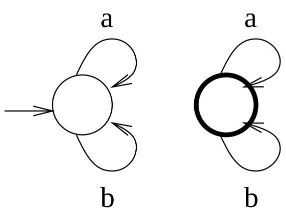

Figure 7.2: Automaton for matching the empty language $\emptyset$

Examples

1. By applying the translation rules above to the automaton of Figure 7.1 we obtain the following RUSSEL program:

```
init_action;
begin
   nar := 0; /* s0 is not accepting */
   trigger off for_next s0;
   trigger off at_completion result
end;

rule s0;
```

```
if  e = 'a' --> begin nar := nar + 1; trigger off for_next s1 end;
    e = 'b' --> begin nar := nar + 1; trigger off for_next s1 end
fi;

rule s1;
begin
   nar := nar - 1;
   if  e = 'a' --> begin nar := nar + 1; trigger off for_next s1 end;
       e = 'b' --> begin nar := nar + 1; trigger off for_next s1 end
   fi
end;

rule result;
if nar > 0 --> print('YES'); true --> print('NO') fi.
```

2. The automaton in Figure 7.2 (representing the empty language $\emptyset$) translates to the program

```
init_action;
begin
   nar := 0; /* s0 is not accepting */
   trigger off for_next s0;
   trigger off at_completion result
end;

rule s0;
if  e = 'a' --> trigger off for_next s0;
    e = 'b' --> trigger off for_next s0
fi;

rule s1;
begin
   nar := nar - 1;
   if  e = 'a' --> begin nar := nar + 1; trigger off for_next s1 end;
       e = 'b' --> begin nar := nar + 1; trigger off for_next s1 end
   fi
end;

rule result;
if nar > 0 --> print('YES'); true --> print('NO') fi.
```

3. The automaton in Figure 7.3 representing the language $\{\epsilon\}$, where ϵ denotes the empty word, translates into the following RUSSEL program:

```
init_action;
begin
   nar := 1; /* s0 is an accepting */
   trigger off for_next s0;
   trigger off at_completion result
end;

rule s0;
begin
   nar := nar - 1;
```

```
      if  e = 'a' --> trigger off for_next s1;
          e = 'b' --> trigger off for_next s1
      fi
   end;

   rule s1;
   if  e = 'a' --> trigger off for_next s1;
       e = 'b' --> trigger off for_next s1
   fi;

   rule result;
   if nar > 0 --> print('YES'); true --> print('NO') fi.
```

We now prove that the RUSSEL program P derived from an automaton (whether deterministic or nondeterministic) M satisfies the following property.

Property 1 *After processing any sequence of records, the value of the variable* `nar` *is equal to the number of accepting rule instances that are active for the next event in the sequence.*

The proof is by induction on the length of the input sequence e_1, ..., e_k.

BASIS $k = 0$, if the start state s_0 is accepting, `nar` is initialized by the init action to 1 and the accepting rule s_0 is the only one that is active for the first record since it is triggered by the init action. Otherwise, the start state s_0 is not accepting and in this case the init action sets `nar` to 0. We verify that there is no accepting rule that is active for the first record since the triggered rule s_0 is the only one rule that is triggered and is not an accepting one.

INDUCTION Suppose that after processing e_k, `nar` is the number of accepting rules that are active for the next record and suppose that after processing e_{k+1}, the number of accepting rules that are active for the next event is l. Because the processing of e_k has triggered l accepting rules, it has incremented `nar` by l. On the other hand, when the `nar` accepting rules are executed against e_{k+1}, each of them has decremented `nar` by 1 so that the new value of nar after processing e_{k+1} is `nar` $+ \ l$ - `nar` $= l$.

Given a RUSSEL program P corresponding to an automaton M, we say that P *accepts* an input sequence (encoded in an audit file) if P prints the message "YES". The input sequence is said to be *rejected* by P if P prints the message "NO".

Based on the above property, we show that the automaton M and the RUSSEL program P it generates using the translation rules have exactly the same language. In other words, the automaton A accepts exactly the same input sequences as the program P. In fact there is a closer connection between what the automaton M does and what the program P does:

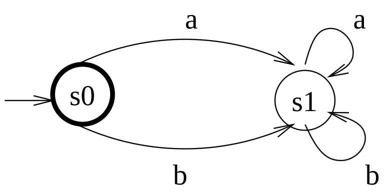

Figure 7.3: Automaton for matching the empty word ϵ

Property 2 *When M is in a state s after reading an input sequence $e_1, \cdots, e_n$ $(n \geq 0)$ then after P has read the same input sequence, the rule which is active for the next record is the rule s associated with the state s by the translation rules.*

We prove this property by induction on k, the length of the so far read input sequence both by the automaton and the corresponding RUSSEL program.

BASIS $k = 0$, the automaton is in the start state s_0 and the input sequence so far read is the empty word ϵ. Just before reading the very first input event, P has already executed the init action, which (by definition of the translation rules) makes active the rule s_0 for the first input event. Since the rule s_0 is associated with the start state s_0, this means that the property holds for $k = 0$.

INDUCTION Suppose the property holds for k and consider an input sequence $e_1, \cdots, e_k, e_{k+1}$. Assuming that after reading the latter sequence the automaton is in state s and has made a sequence of transitions $s1, \cdots, s_k, s$ from the start state s_0 to s, then this path of transitions appears as shown in Figure 7.4; that is M goes through some state s_k just before making the last transition to s on input e_{k+1}. By the induction hypothesis, when M reached the state

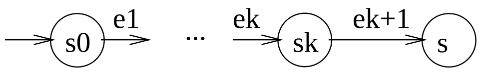

Figure 7.4: Induction step

s_k from its start state s_0 along the path labeled by $e_1, \cdots, e_k$ and after P has read the same input sequence, the rule which is active for the event e_{k+1} is the one associated with state s_k. Since on input e_{k+1} M made a transition to state s, there is a guarded action

```
e = 'e_{k+1}' --> begin [nar := nar + 1]; trigger off for_next s_1 end
```

which makes the rule s active for the next event when executed on e_{k+1}. Thus the property holds for the inductive step.

Now we use Property 2 in order to prove our claim that M and P accept exactly the same input sequences. Let $s_1, \cdots, s_n$ be the states that are visited by M along the path labeled by $e_1, \cdots, e_n$. If M accepts the latter sequence, then s_n is an accepting state. It follows from Property 2 that after P has read the same input sequence, s_n is the rule which is active for the next event, since (by definition of the translation rules) the rule s_n is accepting, then (by Property 1) `nar = 1`. Now because we reached the end of the audit file which encodes the input sequence, `nar` will be equal to 1. Thus the rule *result* will print the message "YES" at completion time.

Since the translation rules are straightforward, in the sense that they only depend on the syntactical structure of the automaton, it is possible to mechanically generate RUSSEL code for matching arbitrary patterns which are describable by finite state automata.

7.4 Nondeterministic Automata

Similar to deterministic automata, we present translation rules for converting a nondeterministic automaton to a RUSSEL program that realizes the matching of the same event pattern. The translation rules presented here generalize the previous ones and are identical except for the translation rule number 3 which becomes

```
begin
   nar := nar - 1;
   if e = 'e1' --> begin [ nar := nar + 1; ] trigger off for_next s1 end fi;

            ⋮

   if e = 'en' --> begin [ nar := nar + 1; ] trigger off for_next sn end fi
end
```

During the execution of the RUSSEL programs obtained in the previous section, there is at most one active rule instance since we only have one `if fi` statement. On the other hand, if we use the translation rules presented above for converting a deterministic automaton, then we still have at most one rule instance which is active at any time because at most one guard expression will evaluate to true. However, in case the automaton is nondeterministic, we may have several active rules at the same time.

Note that Property 1 still holds for RUSSEL programs obtained by using the translation rules for nondeterministic automata since the proof does not assume that the automaton is deterministic. Moreover, the relation of a nondeterministic automaton M to the corresponding RUSSEL program P becomes as follows.

Property 3 *When M is in a set of states $\{s_1, \cdots, s_n\}$ after reading an input sequence say $e_1, \cdots, e_k$, then after P has read the same input sequence, the set of rules in P which are active for the next event are the ones associated with the states $s_1, \cdots, s_n$, according to the translation rules.*

Similarly, this property can be proved by induction on k, the length of the input sequence. However, we will not provide this proof since it is very similar to the one presented for deterministic automata.

Now we (again) similarly use Property 3 and Property 1 in order to prove that M and the corresponding program P accept the same input sequences. Let $\{s_1, \cdots, s_n\}$ be the set of states that are reached by M after being exercised with some input sequence $e_1, \cdots, e_k$. By definition of acceptance in the case of nondeterministic automata, at least one state $s_i \in \{s_1, \cdots, s_n\}$ is an accepting state. It follows from Property 3 that after P has read the same input sequence, $s_1, \cdots, s_n$ are the rules which are active for the next event. Since (by definition of the translation rules) there is at least one rule $s_i \in \{s_1, \cdots, s_n\}$ which is an accepting rule. By Property 1 we have `nar` $\geq$ `1`. Now because we reached the end of the audit file, the rule *result* will print the message "YES", at completion time meaning that P has accepted the input sequence.

7.5 Conclusion

This chapter showed that finite state automata can be systematically translated to RUSSEL programs that realize the matching of the same event patterns as the automata. Moreover, since by Kleen's theorem [39], regular expressions are equivalent to finite state automata, we can first produce a finite state automaton from a given regular expression, and then from that automaton (which is perhaps a nondeterministic one) we can use our translation rules to generate a RUSSEL program that matches the same pattern as the original regular expression. This chapter also showed that nondeterministic algorithms can be expressed naturally in RUSSEL.

Chapter 8

Example Detection Rules

8.1 Introduction

In this chapter we present example RUSSEL programs for detecting intrusion patterns under Solaris 2 Basic Security Module (BSM). We have implemented a format adaptor program for transforming BSM audit files into NADF format and also some miscellaneous C-routines, which are useful for writing RUSSEL programs for Solaris 2 operating system. The presented audit trail analysis programs are used for detecting real-life intrusion patterns in our site. Section 8.2 summarizes the format and meaning of BSM audit records and is a pre-requisite for understanding the example detection programs contained in Section 8.3. The latter section presents four selected intrusion patterns. In each example, we first describe the security vulnerability and how it is exploited to intrude the system; we then characterize the intrusion pattern and provide a RUSSEL program for detecting it in the audit trail. Additional example detection programs are presented in Appendix B. Conventionally, field names appear in uppercase letters and variable names appear in italic. This convention is used throughout the presented examples.

8.2 BSM Audit Records

A detailed description of BSM audit trail format and meaning is beyond the scope of this thesis; the interested reader is referred to [41] for an extensive presentation of BSM auditing features, audit record binary format, and audit installation and configuration procedures. We only emphasize a selected subset of audit record structures, which are relevant to our discussions of the presented examples of Section 8.3.

An audit record under BSM describes the occurrence of a single event and is composed of a sequence of *audit tokens*, each of which describes the attributes of an audited event. The sequence of audit tokens comprising an audit record depends on the event type and includes the time stamp, the process id, the user id. An audit record always begins with a *header* token containing (among other items) the event type and the time and date the event occurred. The *subject* token is also part of every record and contains (among other items) the user id, group id, and process id of the process that caused the occurrence of the event. An *argument* token describes a system call argument and contains the system call argument number and the system call argument value. An *exec_args* token is part of every audit record corresponding to an `exec` system call and is composed of the `exec` system call argument list. A *path* token describes the outcome (success or failure) of the audited event and is composed of the user return error and the system call exit status. When the user return error is 0, this indicates that the event is successful; it indicates failure otherwise. An audit record contains at least the *header*, *subject*, and *return* tokens. Table 8.1 summarizes the above described tokens, where for each token type we provide the list of its fields and the format and meaning of each. In the examples of Section 8.3, we will use the same field names as in this table in order to refer to NADF record fields.

TokenType	*FieldName*	*Format*	*Meaning*
header	HEADER_EVENT	1 byte	event type
	HEADER_TIME	8 bytes	time and date of event
subject	SUBJ_AUID	4 bytes	invariable audit user id
	SUBJ_EUID	4 bytes	effective user id
	SUBJ_EGID	4 bytes	effective group id
	SUBJ_RUID	4 bytes	real user id
	SUBJ_RGID	4 bytes	real group id
	SUBJ_PID	4 bytes	process id
argument	ARG_VAL_1	4 bytes	system call argument
	ARG_VAL_2	4bytes	system call argument
	...	...	...
attribute	ATTR_MODE	4 bytes	file access mode and type
	ATTR_UID	4 bytes	file owner user id
	ATTR_GID	4 bytes	file owner group id
exec_args	EXEC_ARG_1	string	exec system call argument
	EXEC_ARG_2	string	exec system call argument
	...	...	...
path	PATH_NAME	string	object's absolute access path
return	RETURN_ERROR	1 byte	system call return error
	RETURN_STATUS	4 bytes	process return value

Table 8.1: BSM Audit Tokens

8.3 Example RUSSEL Attack Detection Programs

8.3.1 Execution of Known Attack Programs

This attack scenario involves the execution by a malicious user of a known attack program such as:

Crack for cracking encrypted user passwords by trying a number of common passwords and by systematically trying all possible words in a dictionary file.

Tiger, Cops for scanning the system configuration to see if a security vulnerability is found.

ISS, Satan for probing the network services for any security vulnerability.

The attacker runs such programs in order to identify a security hole which s/he then exploits in order to intrude the system. Actually, the execution of these programs does not constitute an intrusion as such, but serves as a preparation phase before launching the actual attack. Therefore, if the execution of any of these programs is detected, this means that the system is about to be under an imminent attack, and hence the execution of these programs should immediately be brought to the attention of the security administrator.

The occurrence of the above scenario is manifested in the audit trail by the existence of an event of type `exec` or `execve`, where the pathname of the executed program is one of the above attack programs.

The RUSSEL program for detecting this attack pattern is depicted in Figure 8.1 and is composed of a single rule *known_attack_program* (besides the init action). This rule examines the current record in order to see if the event is successful (`RETURN_ERROR = 0`), of type `exec` or `execve`, and where the pathname of the executed program matches one of the known attack programs. If such an audit record is found, a message is printed on the standard output indicating the identity of the attacker and the attack program involved. The latter information items about the detected attack are retrieved from the current audit record. Otherwise, if the current record does not satisfy the guard expression, it is simply ignored. In either cases, the rule *known_attack_program* retriggers itself for rest of the audit trail in order to examine subsequent records.

```
rule known_attack_program;
var error, event:  integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and (event = 7 or event = 23) and /* exec(2) or execve(2) */
      (match('.*cops.*', PATH_NAME) = 1 or
      match('.*satan.*', PATH_NAME) = 1 or
      match('.*tiger.*', PATH_NAME) = 1 or
      match('.*crack.*', PATH_NAME) = 1)
      -->
      begin
         print('User id ', strToInt(SUBJ_AUID));
         println(' has successfully executed ', PATH_NAME)
      end
   fi;
   trigger off for_next known_attack_program
end;

init_action;
begin
   trigger off for_next known_attack_program
end.
```

Figure 8.1: Execution of a Known Attack Program

Note that in order to optimize guard expression evaluation, the less costly conditions appear first. In particular, if the reported action fails, the other conditions are not evaluated at all. We make use of this optimization in the rest of this chapter whenever possible.

Recall that in RUSSEL all NADF audit record fields are considered as uninterpreted string of bytes. Consequently, conversion routines must be used if for instance one needs to interpret a byte string as the binary representation of a number. The predefined C-routine $strToInt()$ receives a byte string of length 1, 2, or 4 and returns the corresponding integer number. For example, the audit data fields, `HEADER_EVENT, HEADER_TIME, ATTR_MODE` must be converted using $strToInt()$ if they appear in arithmetic expressions; otherwise, this would result in a type mismatch. However, for the sake of simplicity and throughout the remainder of this chapter[1], we intentionally omit the use of this conversion routine in order to clarify the code.

8.3.2 Overwriting Files Outside the Spool Directory

The purpose of this attack is to force the line printer program `/user/lib/lpr` into overwriting an arbitrary file in the file system. The vulnerability exists because it is very easy to predict the names of the files to be enqueued in the spool directory `/var/spool/lpr/printer`. When *lpr* is requested to print a file, it stores a copy of the file in the spool directory under a name of the form:

`dfA274pascal.info.fundp.ac.be`

i.e., the characters `dfA`, followed by a 3 digit number, and followed by the IP address of the machine the print request originated from. The 3 digit number is incremented for each new print request and wraps around upon reaching 999, and this allows an attacker to predict the file names used by *lpr*. Additionally, when *lpr* is given the option -s, it creates a link to the file to be printed instead of making a fresh copy of the file. The exploitation works as follows. First, the attacker creates a link, say `x`, to the file s/he wants to overwrite and then issues a request to print file `x` with option `-s`. Afterwards, s/he makes a sufficient number of print requests until the names used by *lpr* wrap around. A that time *lpr* will overwrite the file `x`, provided it is still in the spool directory. However, since the file `x` is a link, *lpr* will follow the link, thereby overwriting the destination of the link. In practice, the attacker would use the option -q which causes *lpr* to store a file into the spool directory without printing it until it is later instructed to do so. This option is used by the attacker to hide the high number of print requests which causes the names used by *lpr* to wrap around. This attack could be used in order to overwrite the password file with one of the attacker's choosing.

An attack signature which is useful for detecting the above scenario is that *lpr* overwrites a file outside the spool directory because *lpr* is not supposed to create any file outside that directory. The attack pattern is composed of two events. The first event is the execution of *lpr* which produces an `exec` or `execve` event with the pathname of *lpr* as an audit data field. The second event is that the initiated *lpr* process subsequently produces an event of type `write`, where the created file is outside the spool directory.

To detect the above event pattern, we provide a RUSSEL rule for detecting the occurrence of an `exec` or `execve` record, where the pathname `PATH_NAME` of the executed program is that of *lpr*. Upon detecting such an event, *exec_lpr* extracts the process id and real user id from the current record and triggers off the rule *lpr_creat* with the process id and effective user id as arguments. The latter rule then seeks out in the rest of the audit trail the occurrence of a successful `creat` event by the same process id as its parameter and where the created file is outside the spool directory. The resultant RUSSEL program is depicted in Figure 8.2.

[1]Except for the examples presented in Appendix B.

```
rule exec_lpr;
begin
   if
      RETURN_ERROR = 0
      and (HEADER_EVENT = 7 or HEADER_EVENT = 23) /* exec(2) or execve(2) */
      and match('.*lpr.*', PATH_NAME) = 1 and SUBJ_EUID = 0
      -->
      trigger off for_next exec_lpr(SUBJ_PID, PATH_NAME, SUBJ_RUID)
   fi;
   trigger off for_next exec_lpr
end;

rule lpr_creat(pid:  integer; prog:  string; ruid:  integer);
if
   RETURN_ERROR = 0 and SUBJ_PID = pid and HEADER_EVENT = 4 /* creat(2) */
   -->
   if
      isPrefix(prog, '/var/spool/') = 0
      -->
      println('User id ', ruid, ' attempted to copy over file ', PATH_NAME)
   fi;
   true
   -->
   trigger off for_next lpr_creat(pid, prog, ruid)
fi;

init_action;
begin
   trigger off for_next exec_lpr
end.
```

Figure 8.2: Overwriting Files outside the Spool Directory

8.3.3 Illegal Read Access to a Device File

UNIX views all devices (such as disks, tapes, the keyboard, the screen, etc.) as special files on which I/O operations can be performed transparently as if they were common files on the file system. Although a file may be adequately protected against read access, a malicious user can still read it directly from the disk (using, for example, *cat*). Similarly, the device files /dev/drum (the swapper), /dev/mem (user memory space), /dev/kmem (kernel memory space), /dev/le (the Ethernet interface), can be displayed for reading kernel or user data in main memory or monitor packet traffic on the network. For instance, by reading /dev/mem, a malicious user could view data buffers in memory and discover the content of files being edited by another user. Consequently, this type of device files should deny read access to everyone but root. Improperly protecting these device files against unauthorized access could lead to the disclosure of sensitive information. Moreover, the superuser himself is usually not expected to read directly from these files since only trusted system programs (such as ps, etc.) are normally supposed to read from these files.

The intrusion pattern yielded by such a malicious behavior is composed of two events. The first event is the creation of a process through the `exec` or `execve` system call. The second event in the pattern is of type `read` and is performed by the same process id as the previously created process.

```
rule exec;
begin
   fi
      RETURN_ERROR = 0 and (HEADER_EVENT = 7 or HEADER_EVENT = 23) /* exec(2) or execve(2) */
      -->
      trigger off for_next illegal_read(PATH_NAME, SUBJ_PID)
   fi;
   trigger off for_next exec
end;

rule illegal_read(progr: string; pid: integer);
if
   RETURN_ERROR = 0 and pid = SUBJ_PID and HEADER_EVENT = 1 /* exit() */
   --> skip;
   /* AUE_OPEN_R or AUE_OPEN_RC or ...  or OPEN_RWTC */
   (HEADER_EVENT = 72 or HEADER_EVENT = 73 or HEADER_EVENT = 74 or
    HEADER_EVENT = 75 or HEADER_EVENT = 80 or HEADER_EVENT = 81 or
    HEADER_EVENT = 82 or HEADER_EVENT = 83) /* read access */
   and SUBJ_PID = pid /* same process id */
   and match('.*/devices/.*mem.*', PATH_NAME) = 1
   and isTrusted(progr) = 0 /* not a trusted program */
   -->
   begin
      println('Illegal read access by uid ', SUBJ_RUID, ' Pid ', pid);
      println('To Restricted read access file ', PATH_NAME, ' Command used was ', progr)
   end;
   true
   -->
   trigger off for_next illegal_read(progr, pid)
fi;

init_action;
begin
   trigger off for_next exec
end.
```

Figure 8.3: Illegal Read Access to a Device File

Additionally, the occurrence of this pattern must not involve a trusted system program. As a result, the detection of the above pattern is performed by two detection rules: *exec* and *illegal_read*. The former detects all spawned processes and triggers off the latter with the pathname of the invoked program and the corresponding process id as parameters. *illegal_read* then examines all read operations performed by the process id; whenever a read access is performed on any one of the devices files, an alarm message is printed unless the program performing the read operation is a trusted program. The external C- routine *isTrusted*() is used in order to verify if a program is actually one of the trusted programs. Furthermore, the rule *illegal_read* dies as soon as it detects that the monitored process exits; otherwise, the rule may monitor a process which has already disappeared. The resultant RUSSEL program implementing this analysis is depicted in Figure 8.3. Note that since the pathname of the invoked program is only contained in the `exec` or `execve` event and is not recorded in the subsequent read operations, we need to first detect the occurrence of the `exec` or `execve` events in or order to extract the pathname. Note also that the less costly conditions are placed at the front of the guard expression to provide optimized evaluation of the guard.

8.4 Mail Vulnerability Exploitation

This section presents how the penetration scenario described in the chapter about related works can be detected using the language RUSSEL. We provide five versions of a RUSSEL program for detecting this penetration in order to illustrate the capabilities of the language.

8.4.1 Total order

The first version assumes that the attacker types the four commands:

```
cp /bin/sh /usr/spool/mail/root
chmod 4755 /usr/spool/mail/root
touch x
mail root < x
```

in sequence and so it does not take into account the fact that the `touch` command may occur before `cp` or before `chmod`. Consequently, the purpose of the first version is to detect the totally ordered sequence of commands `(cp, chmod, touch, mail)`. The command `cp` yields a `creat` system call, the command `chmod` yields a `chmod` system call, the command `touch` yields a `stat` system call followed by a `utimes` system call, and the command `mail` yields an `exec` or `execve` system call. The resulting penetration signature is therefore composed of the audit record sequence `(creat, chmod, stat, utimes, exec` or `execve)`. The audit records comprising the latter sequence are related as follows. The `creat` and `chmod` system calls must be performed on the file `/var/spool/mail/root`, the `stat` and `utimes` system calls must be performed on the same file, and the `exec` or `execve` system call must have three arguments: the path name of the mail command, the recipient user name which is *root*, and the file involved in `stat` and `utimes` system calls. Moreover, we do not require that the audit records comprising this sequence immediately follow each other, so that they may possibly be separated by intervening audit records because the auditing mechanism generates audit records asynchronously for all processes in the system.

The RUSSEL program for detecting the above audit record sequence is composed of five rules: *create*, *chmod*, *stat*, *utimes*, and *exec*; each rule being responsible for detecting a particular audit record in the sequence. When a rule finds that its corresponding event matches the current record, it triggers off for the next record the rule which is responsible for detecting the next event in the sequence. Otherwise, it retriggers itself for the next record. When the rule *exec* encounters an audit record satisfying its corresponding event, it reports the detected penetration scenario using the involved process id and user effective user id, which are contained in the current record. The resulting program is depicted in Figures 8.4 and 8.5. The predefined C-routine *issetuid*() which is used in the rule *chmod* returns 1 if its permission mode argument has the set user id bit enabled and 0 otherwise. Note in particular that the path name of the file involved in the `stat` system call is passed to the rule *utimes* and then to the rule *exec*. The rule *utimes* verifies that the detected `utimes` system call involves the same file as its parameter and similarly, the rule *exec* verifies that the detected execution of *mail* uses the same file as its parameter.

```
rule create;
if
  RETURN_ERROR = 0 and HEADER_EVENT = 4 /* creat(2) */
  and PATH_NAME = '/usr/spool/mail/root'
  --> trigger off for_next chmod;
  true
  -->
  trigger off for_next create
fi;

rule chmod;
if
  RETURN_ERROR = 0 and HEADER_EVENT = 10 /* chmod(2) */
  and PATH_NAME = '/usr/spool/mail/root'
  and issetuid(ARG_VAL_1) = 1 /* setuid enabled */
  --> trigger off for_next stat;
  true
  -->
  trigger off for_next chmod
fi;
rule stat;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 16 or HEADER_EVENT = 23) and SUBJ_EUID != 0
  --> trigger off for_next utimes(PATH_NAME);
  true
  -->
  trigger off for_next stat
fi;

rule utimes(path:  string);
if
  RETURN_ERROR = 0 and HEADER_EVENT = 49 and PATH_NAME = path
  -->
  trigger off for_next exec(path);
  true --> trigger off for_next utimes(path)
fi;
```

Figure 8.4: Detecting Mail Exploit (Version 0)

```
rule exec(path: string;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 7 or HEADER_EVENT = 23) and file != ''
  and match('.*mail.*', ARG_VAL_1) = 1 /* exec mail */
  and match('.*root.*', ARG_VAL_2) = 1 /* recipient is root */
  and path = ARG_VAL_3 = 1 /* mail file */
  -->
  begin
    print('Root setid shell created through mail exploit ');
    println('Pid ', SUBJ_PID, ' Euid ', SUBJ_EUID)
  end;
  true
  -->
  trigger off for_next exec(path)
fi;

init_action;
  trigger off for_next create.
```

Figure 8.5: Detecting Mail Exploit (Version 0) (Continued)

8.4.2 Partial order

The second version is a straightforward encoding of the automaton depicted in Figure 2.4 into a RUSSEL program in order to take into account the partial order of the events comprising the sequence. We use five rules to detect the scenario, each of which being responsible for detecting the occurrence of a particular event satisfying the corresponding guard in the automaton. Moreover, the synchronization point is implemented by means of the global variable `file`, which indicates whether the sequence `(stat, utimes)` has occurred. When the sequence `(creat, chmod)` is detected, the rule *exec_mail* is triggered and remains active until the global variable `file` is set to some file name, meaning that `touch` has occurred. As long as the sequence `(stat, utimes)` has not been found yet, the variable `file` is set to the empty string. As soon as the rule *exec_mail* finds that `file` is not set to the empty string, it seeks out in the rest of the audit trail the occurrence of an `exec` or `execve` event, where the executed program is *mail*, `file` is the argument of *mail*(), and the mail recipient is *root*. Note that the variable `file` plays two roles: it serves as an indication of the synchronization and also as the name of the `touch`ed file.

8.4.3 Partial order using temporal window

In practice, a rule which monitors the activity of a given process should stop as soon as it detects an `exit` system call by the same process. Moreover, in certain cases the audit record pattern yielded by a given process usually falls within a short time interval. This technique is referred to as the *temporal window technique*. In our case the two consecutive system calls `stat` and `utimes` generated by `touch` are produced by the same process id and fall within a very short temporal window. This observation gives us a clue for improving the efficiency of the previous version. On the one hand, as soon as the process which performed the `stat` system call dies by calling the `exit` system call, *utimes* restarts the search for a new sequence `(stat, utimes)` by retriggering the rule *stat* over again. On the other hand, if *utimes* does not encounter an `utimes` system call within a

```
global file: string;
rule stat;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 16 or HEADER_EVENT = 23) and SUBJ_EUID != 0
  --> trigger off for_next utimes(PATH_NAME);
  true
  -->
  trigger off for_next stat
fi;

rule utimes(path: string);
if
  RETURN_ERROR = 0 and HEADER_EVENT = 49 and PATH_NAME = path
  -->
  file := path;
  true --> trigger off for_next utimes(path)
fi;

rule create;
if
  RETURN_ERROR = 0 and HEADER_EVENT = 4 /* creat(2) */
  and PATH_NAME = '/usr/spool/mail/root'
  --> trigger off for_next chmod;
  true
  -->
  trigger off for_next create
fi;

rule chmod;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 10 or HEADER_EVENT = 39) /* chmod(2) or fchmod(2) */
  and PATH_NAME = '/usr/spool/mail/root'
  and issetuid(ARG_VAL_1) = 1 /* setuid enabled */
  --> trigger off for_next exec_mail;
  true
  -->
  trigger off for_next chmod
fi;
```

Figure 8.6: Detecting Mail Exploit (Version 1)

```
rule exec_mail;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 7 or HEADER_EVENT = 23) and file != ''
  and match('.*mail.*', ARG_VAL_1) = 1 /* exec mail */
  and match('.*root.*', ARG_VAL_2) = 1 /* recipient is root */
  and file = ARG_VAL_3 = 1 /* mail file */
  -->
  begin
    print('Root setid shell created through mail exploit ');
    println('Pid ', SUBJ_PID, ' Euid ', SUBJ_EUID)
  end;
  true
  -->
  trigger off for_next exec_mail
fi;

init_action;
begin
  file := '';
  trigger off for_next stat;
  trigger off for_next create
end.
```

Figure 8.7: Detecting Mail Exploit (Version 1) (Continued)

given time period, it must expire because it is very unlikely that the `stat` system call corresponds to a `touch` command. Similar to the case where the process exits without calling `utimes`, the rule *utimes* retriggers off the rule *stat* in order to search for a new sequence `(stat, utimes)`. The modified version of the rules *stat* and *utimes* are depicted in Figure 8.8.

8.4.4 Temporal locality

In the previous version, the two pairs of rules (*stat*, *utimes*) and (*create*, *chmod*) are completely asynchronous whereas in reality, the attacker would rather try to perform the sequence `(creat, chmod)` first, because the latter sequence is more difficult to achieve. For instance, the attacker is unlikely to `touch` a bogus file and then waits a day or two for the root mail box to become empty. S/he would rather execute the `touch` command within a limited time period before or after the occurrence of the sequence `(creat, chmod)`. Consequently, we make use of the temporal window technique to further improve the previous version. The basic idea is to consider a successfully detected sequence `(stat, utimes)` obsolete after a given time period after which the search for a new sequence is started over again. We utilize the additional global variable *time*, which is the time stamp of the detected sequence `(stat, utimes)`[2]. When the rule *utimes* detects the `utimes` system call, it stores in $file$ and *time* the corresponding file name and time stamp, respectively. The rule *utimes* then triggers off another rule *touch* with an expiration time as argument. The role of the latter rule is to reset the variables $file$ and *time* upon encountering a record with a time stamp beyond the expiration time, and then retriggers the rule *stat* in order to search for a new `(stat, utimes)` sequence. So long as the expiration time is not reached yet, the rule *touch*

[2]Actually the time stamp of the `utimes` system call in the detected sequence.

```
global file: string;
rule stat;
if
  RETURN_ERROR = 0 and (HEADER_EVENT = 16 or HEADER_EVENT = 23) and SUBJ_EUID != 0
  --> trigger off for_next utimes(PATH_NAME, SUBJ_PID, HEADER_TIME + 300);
  true
  -->
  trigger off for_next stat
fi;

rule utimes(path: string, pid, expiration: integer);
if
  RETURN_ERROR = 0 and HEADER_EVENT = 49 /* utimes(2) */
  and PATH_NAME = path
  -->
  file := path;
  (RETURN_ERROR = 0 and HEADER_EVENT = 1) /* exit(2) */
  or timestamp > expiration
  -->
  begin
    file := '';
    trigger off for_next stat
  end;
  true
  -->
  trigger off for_next utimes(path, pid, expiration)
fi;
```

Figure 8.8: Detecting Mail Exploit (Version 2)

simply retriggers itself for the next record. The modified versions of *stat* and *utimes* as well as the rule *touch* are shown in Figure 8.9.

8.4.5 A general pattern

As mentioned in Section 2.6.2, the `touch` command is not vital to the successful completion of the attack scenario and it is sufficient to detect the sequence `(creat, chmod, chown)`. The `chown` system call is yielded (among other system calls) by the execution of the *mail* delivery program. However, we must ensure that the set user id permission bit of `/usr/spool/mail/root` is still enabled at the time the system call `chown` occurs; otherwise the attack would not be successful. The parameter $flag$ (see Figures 8.10 and 8.11) of the rule *chownmod* is set to 1 or 0 if the most recent `chmod` event had enabled or cleared the set user id permission bit, respectively. If $flag = 0$, *chownmod* searches for the occurrence of a `chmod` event enabling the set user id bit, in which case it retriggers itself with $flag = 1$; otherwise, it retriggers itself without changing $flag$. If $flag = 1$, *chownmod* searches for a `chown` event, which changes the ownership of `/usr/spool/mail/root` to root. In the latter case, a successful attack has been detected and a message is printed to indicate the occurrence of the attack. Additionally, if $flag = 1$ and a `chmod` event disables the set user id bit, the rule *chownmod* retriggers itself with $flag = 0$. It is worth noting that if we ignored the fact that the set user id permission bit can possibly be cleared before the occurrence of the `chown` event, the program would raise a false alarm because `/usr/spool/mail/root` may not be set user id enabled.

```
global file: string;
global time: integer;
rule utimes(path: string, pid, expiration: integer);
if
  RETURN_ERROR = 0 and HEADER_EVENT = 49 /* utimes(2) */
  and PATH_NAME = path
  -->
  begin
    file := path;
    time := HEADER_TIME;
    trigger off for_next touch(HEADER_TIME + 600)
  end;
  (RETURN_ERROR = 0 and HEADER_EVENT = 1) /* exit(2) */
  or timestamp > expiration
  -->
  begin
    file := '';
    time := 0;
    trigger off for_next stat
  end;
  true
  -->
  trigger off for_next utimes(path, pid, expiration)
fi;

rule touch(expiration: integer);
if
  HEADER_TIME > expiration
  -->
  begin
    file := '';
    time := 0;
    trigger off for_next stat
  end;
  true
  -->
  trigger off for_next touch(expiration)
fi;
```

Figure 8.9: Detecting Mail Exploit (Version 3)

```
rule create;
if
   RETURN_ERROR = 0 and HEADER_EVENT = 4 /* creat(2) */
   and PATH_NAME = '/usr/spool/mail/root'
   --> trigger off for_next chownmod(0);
   true
   -->
   trigger off for_next create
fi;

init_action;
begin
   trigger off for_next create
end.
```

Figure 8.10: Detecting Mail Exploit (Version 4)

```
rule chownmod(flag:  integer);
if
   flag = 0
   -->
   if
      RETURN_ERROR = 0 and (HEADER_EVENT = 10 or HEADER_EVENT = 39) /* chmod(2) or fchmod(2) */
      and PATH_NAME = '/usr/spool/mail/root' and issetuid(ARG_VAL_1) = 1
      -->
      trigger off for_next chownmod(1)
   fi;
   true
   -->
   trigger off for_next chownmod(flag)
   fi;
   flag = 1
   -->
   if
      RETURN_ERROR = 0 and (HEADER_EVENT = 10 or HEADER_EVENT = 39) /* chmod(2) or fchmod(2) */
      and PATH_NAME = '/usr/spool/mail/root' and issetuid(ARG_VAL_1) = 0
      -->
      trigger off for_next chownmod(0);
      RETURN_ERROR = 0 and (HEADER_EVENT = 11 or HEADER_EVENT = 38) /* chown(2) or fchown(2) */
      and PATH_NAME = '/usr/spool/mail/root' and ARG_VAL_1 = 0
      -->
      begin
         print('Root setid shell created through mail exploit ');
         println('Pid ', SUBJ_PID, ' Ruid ', SUBJ_RUID)
      end
   fi;
   true
   -->
   trigger off for_next chownmod(flag)
fi;
```

Figure 8.11: Detecting Mail Exploit (Version 4 continued)

Chapter 9

Integrating IDS and Configuration Analysis

9.1 Introduction

The occurrence of an intrusion is generally due to the presence of a security vulnerability, which is exploited by an attacker in order to compromise the system security. A security vulnerability may result from logical design flaws or programming errors in the protection software itself or may be due to inadequate configuration of the protection mechanism by the security administrator. Indeed, the process of configuring the protection mechanism involves choosing between many options, which may have unexpected interactions. Consequently, adequate protection configuration is a challenging task for unexperienced security administrators. There are several tools that were created for the purpose of testing an existing configuration in an attempt to uncover common configuration problems, which would constitute security vulnerabilities. The use of such tools — henceforth referred to as configuration analysis systems — provide a preventive means of reducing the risk of introducing vulnerabilities that imperil security.

The purpose of this chapter is to extend ASAX with a configuration analysis component, based on the observation that intrusion detection can be more efficient by taking advantage of the knowledge about monitored system's vulnerability to attacks. Combining intrusion detection and configuration analysis allows us to achieve a better tuning of intrusion detection since the system has only to enable detection rules that are specifically required by the current state of the configuration. We present a system especially designed for continuously checking target-system vulnerability to attacks, which is coupled with ASAX, thereby enabling a dynamically adaptive detection rule set. The configuration analysis component is based on a datalog deductive language, which is useful for modeling the current security level of a UNIX configuration, where the fact base is incrementally updated by analyzing audit trails in real-time.

The rest of this chapter is organized as follows. Section 9.2 examines the challenges related to adequate configuration of the protection system. Section 9.3 reviews related works on configuration analysis. Section 9.4 outlines the benefits of integrating intrusion detection with configuration analysis, defines the novel deductive language component, and presents the architecture of the integrated system. Section 9.5 contains detailed examples demonstrating the functionalities of the integrated system. In Section 9.6, we focus on the implementation issues related to the fact base initialization an incremental update. Section 9.7 describes the command line language of the integrated system. In Section 9.8 we present a performance evaluation of the integrated system that bears out the feasibility of the integrated approach in a real computing environment. Finally, Section 9.9 draws conclusions.

9.2 Protection Configuration Challenges

The UNIX protection model is based on well known concepts such as passwords, files, users, user groups, trusted hosts, exported file systems, etc. The effectiveness of this protection model is dependent not only on the integrity of the protection software and of its related data bases, but also on the expertise of the security administrators in adequately configuring the protection mechanisms to accommodate site-specific security goals. In practice, there are unexpected interactions between seemingly independent configuration decisions, which in the worst case may collapse the entire protection system and let the door open for potential security compromises. Unexpected interaction between configuration decisions mainly stems from the fact that a user may be a member of several groups at the same time. Although this feature (among other features such as trusted hosts) allows UNIX users to share files of common interest in a flexible manner, its adequate control quickly grows in complexity, because the security and integrity of so shared files depend on configuration decisions that are taken by all these group members. For example, consider the case where a user who is a member of two different groups inadvertently sets the permission modes of his `.login` file, thereby allowing an attacker to plant a trojan horse in this startup file. An attacker who successfully compromised this user has access not only to the files owned by that user, but also to all files that are shared among the two groups to which he belongs. Since user groups usually have different levels of privileges, compromising the account of a less privileged user can possibly endanger the security of higher privileged users or groups of users, and in the worst case may even compromise the super-user account itself. Consequently, for a user to decide whether he should share a file with a group, say `friends`, he must — in principle — know to which other groups each member of `friends` belongs to, and recursively. Similar interactions between configuration decisions arise in case a host uses the trust mechanism among a set of machines through `.rhosts` and `/etc/hosts.equiv` files. Since current computing environments are composed of personal workstations, which are administered by unexperienced users, a configuration mistake in any host sharing trust with some other hosts could imperil the security of all these hosts. Thus, it becomes apparent that adequately configuring a protection system is a difficult task in which a single critical misconfiguration could have the effect of collapsing the entire access control in place.

9.3 Protection Configuration Analysis Systems

Configuration analysis systems perform a systematic analysis of the computer configuration in order to uncover possible security breaches that can make the system vulnerable to break-ins, misuse, or abuse. For instance, access control mechanisms may be useless if security critical files or directories contain vulnerabilities or are not adequately protected against malicious corruption. Several configuration analysis tools have been developed and are used periodically to assess the system vulnerability to attacks. For example, the computer oracle password and security system (COPS) [42] reports a list of common vulnerabilities such as weak passwords, world writable user home directories, world writable security-relevant files and directories, unexpected setUID files, improper exports of file systems, etc. A major component of COPS is the SU-KUANG [43] system which is a rule-based security checker system that adopts an attacker's reasoning in searching for security holes in the protection configuration. The administrator specifies an attacker goal (e.g., obtain root access to the system) and a set of initial privileges that are granted to the attacker. Using simple backward chaining, it recursively expands the attacker's goal into one or more subgoals. If the attacker's goal can be eventually reduced to a set of basic facts about the assumed privileges and the actual configuration, a potential attack is found. Moreover, the sequence of actions leading to the attacker's goal are reported. This simple rule-based system has proven very effective in finding security breaches in a Unix configuration and is periodically ran by many administrators to check their systems. NetKuang [44] is a multi-host configuration vulnerability checker that extends the functionalities of SU-KUANG [43] to a network environment. The security administrator tool for analyzing networks (SATAN) [45] and the Internet scanner [46] have been developed to aid in identifying vulnerable network services in a network domain. Other similar systems have also been developed such as Miro [47] and Tripwire [48].

9.4 Integrating Configuration Analysis and Intrusion Detection

9.4.1 Rationales of the Integrated System

The need for integrating intrusion detection and configuration analysis into a single unified system is motivated by two important factors.

First, although existing configuration analysis systems such as COPS and SU-KUNAG allow security administrators to validate configuration decisions, they are designed to be executed only periodically, say every night or every time an important modification of the current configuration is performed. This very periodical nature of existing configuration analysis systems constitutes a limiting factor, because if for some reasons a security weakness is introduced in the protection system in between two successive analyses, an attacker could still exploit this weakness to intrude the system. Moreover, the attacker could create a back-door for future break-ins and cover or correct the introduced weakness so that the next execution of the configuration analysis system may not be able to report it. Alternatively, the attacker can corrupt the configuration analysis system itself so that all subsequent vulnerabilities are not detected, thereby giving the security administrator a false sense of security if he solely relies on the configuration analysis system. Based on this observation, we believe that there is a real need for achieving a continuous analysis, where configuration decisions are tracked in an on-line basis as the system evolves over time. Moreover continuous configuration analysis allows the security administrator to take timely corrective measures. Continuous assessment of the integrity of the protection system is also motivated by the fact that current computing environments evolve more rapidly than before, even for modest-sized user communities.

Second, an intrusion detection system may look for attack scenarios that are irrelevant, because the current configuration is in fact not vulnerable to these attacks. However, we assume that the security administrator is more concerned with detecting successful intrusion attempts than failed ones. The processing resources thus spared by the intrusion detection process must be balanced with the need for detecting both successful and unsuccessful intrusion attempts. Consequently, an intrusion detection tool that integrates both penetration detection and continuous tracking of the degree of system vulnerability to attacks allows security administrators to use a single tool in a consistent manner.

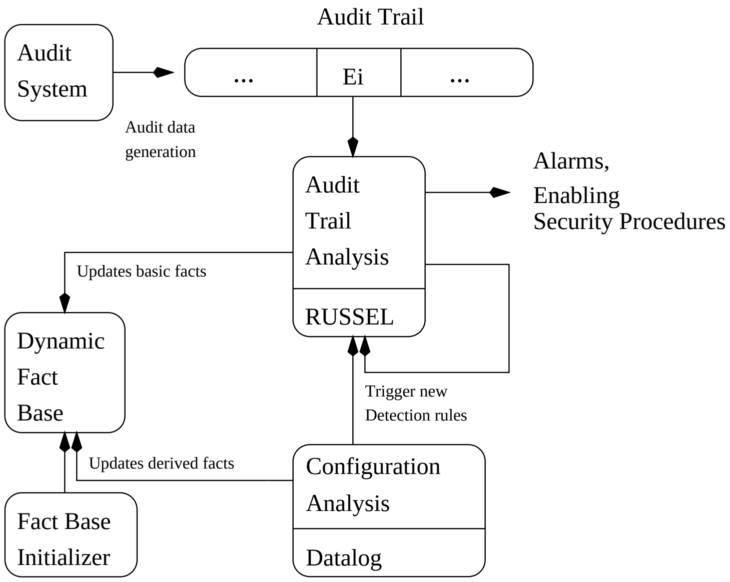

Figure 9.1: Combining Audit Trail Analysis and Configuration Analysis

9.4.2 Overview of the Integrated System (see Figure 9.1)

Upon receiving an event from the audit subsystem, the intrusion detection subsystem executes all currently active detection rules. If the current audit record represents a malicious action, these rules may send alarm messages to alert the security administrator. If the current audit record reports a modification of some security sensitive file, one of the current rules executes a routine that accordingly updates the fact base describing the access control configuration. Additionally, the current rules may trigger new rules for the next record. Afterwards, the configuration analysis subsystem *incrementally* updates the current fact base to accommodate all logical consequences of the last analyzed event. This is done in two steps. First, since the current event may imply that some access rights to security sensitive files are retracted, the corresponding facts are retracted from the fact base as well as all derived facts implied by the deductive rules. Second — and similarly —, as new access rights may be granted for some files, all corresponding basic and derived facts are added to the fact base. Finally, as a consequence of the modifications to the fact base, the intrusion detection component may deactivate irrelevant detection rules and may trigger off some other ones in order to watch for possible users exploiting any newly created breach.

Initialization of the integrated system is performed by the fact base initializer, which builds the fact base reflecting the start-up configuration.

In the next sections, we describe our novel configuration analysis component and the architecture of coupling it with ASAX. Detailed examples are provided in order to illustrate the capabilities of the integrated system.

9.4.3 Informal Description of the Bottom-up Deductive Language

The goal of the deductive language for configuration analysis is to provide a simple formalism for representing the reasoning of an attacker trying to find holes in the configuration. The benefit of the deductive language is an increased flexibility in building and maintaining the attacker reasoning using predicate (Horn clause) logic.

Our language is based on a restricted (datalog like) form of predicate logic and allows us to express how an attacker is able to extend his privileges by exploiting weaknesses in the protection configuration. Attacker goals are represented by atoms (e.g., stealing the identity of user `joe` by an attacker `attacker` is simply represented by the atom `become(attacker, joe)`). There are two kinds of predicates. Derived predicates correspond to an attacker goal such as `become(U, root)` or `write(User, /etc/passwd)`. Basic predicates describe the state of the configuration in terms of access permission modes of files, directory structure, file ownership, etc (e.g., `worldWrite(/home/users/amo)`). Table 9.1 depicts a list of basic facts currently used and their meaning. In this table *F, D, U* and *G* refers to a file, a directory, a user and a group, respectively. A program in our language is a set of rules (see Figure 9.6).

Basic Fact	*Meaning*
worldWrite(F)	F is writable by any user
groupWrite(F, G)	F is writable by the members of G
worldRead(F)	F is readable by any user
groupRead(F, G)	F is readable by the members of G
groupMember(U, G)	U is a member of G
parentDir(D, F)	D is the parent directory of F
start_up(U, F)	F is a start-up file of U
owner(U, F)	F is a owned by U

Table 9.1: Basic Facts

9.4.4 Syntax and Semantics of the Deductive Language

The syntax of the datalog-like language that we use for modeling the attackers' reasoning is defined as follows. An atom is of the form $p(arg_1, ..., arg_n)$ where p is a predicate symbol and $arg_1, ..., arg_n$ are (restricted) terms. A term is either a variable name (starting with an upper case letter, as in Prolog) or a constant. For our specific purpose, constants are interpreted as being either a user name, a group name, or a path name. A rule is of the form h :- $a_1, ..., a_n$ where $h, a_1, ..., a_n$ denote atoms. h is said to be the head of the rule. A program is a set of rules. There are two kinds of predicates: a predicate is said to *derived* if there is at least one rule in the program whose head has the same name. Otherwise, it is called a *basic* predicate. For the sake of efficiency, we also use some built-in predicates. A fact is derived (resp. basic) if its predicate is derived (resp. basic). A deduction is an instance of a rule, i.e., an object of the form $r\sigma$, where r is a rule and σ is a substitution. The union of basic and derived facts is referred to as the *fact base*.

Given a set of basic facts *BF* and a set of rules *SR*, we simultaneously define the set of deductions *SD* and the set of derived facts *DF*, corresponding to *BF* and *SR* as the smallest sets such that

$$\left.\begin{array}{l} r \in SR, \\ r \equiv h : -a_1, ..., a_n, \\ f_1, ..., f_n \in BF \cup DF, \\ \sigma = mgu\{a_1 = f_1, ..., a_n = f_n\} \end{array}\right\} \Rightarrow \begin{array}{l} h\sigma \in DF \textit{ and} \\ r\sigma \in SD \end{array}$$

and we note ⟨SD, DF⟩ = Ded (BF, SR). Given a set of basic facts *BF*, a set of derived facts *DF* and ⟨SD, DF⟩ = Ded(BF, SR), the purpose is to compute the new pair Ded((BF $\setminus\Delta^-) \cup \Delta^+$, SR) where Δ^- (resp. Δ^+) denotes a set of retracted (resp. added) facts. However, instead of computing Ded((BF$\setminus\Delta^-)\cup\Delta^+$, SR) from scratch, we proceed incrementally by reusing the sets *SD* and *DF* we calculated before. This is done in two steps. First, we compute (SD$^-$, DF$^-$) = Ded(*BF* $\setminus\Delta^-$, *SR*) from Δ^-, *SD*, *DF*, *BF* and *SR*. Then we compute Ded((BF$\setminus\Delta^-) \cup \Delta^+$, SR) from Δ^+, SD$^-$, DF$^-$, BF$\setminus\Delta^-$ and SR.

In the following, we say that a fact *f* *contributes* to a deduction $r\sigma$ if $r\sigma$ is of the form

$$h\sigma \text{ :- } a_1\sigma,\ \ldots, a_{i-1}\sigma,\ f\sigma,\ a_{i+1}\sigma,\ \ldots,\ a_n\sigma.$$

During the first step, we remove all deductions to which some removed fact contribute. Simultaneously, we remove all derived facts that are no longer implied by any deduction. (Note that a fact can be implied by several different deductions.) During the second step, we use an incremental fixpoint algorithm (see [49, 50]) to compute all newly derived facts; additionally, we store the corresponding deductions. Given a (initial) fact base B_0 and the fact base B_1 resulting from incrementally updating B_0 as described above, the set of facts $B_1 \setminus B_0$, denoted by B^+, is composed of all facts which are newly added to the fact base. Similarly, the set of facts $B_0 \setminus B_1$, denoted by B^-, is composed of all facts which were removed from the fact base.

```
Retract_ded(Δ⁻, SD, DF, BF, SR)
begin
    Δ := Δ⁻;
    while (Δ ≠ ∅) do
    begin
        Δ := Δ \ {f};
        Sup_ded := list_ded(f);
        while (Sup_ded ≠ ∅) do
        begin
            Sup_ded := Sup_ded \ {d};
            f' := Fact(d);
            SD := SD \ {d};
            if (Nb_ded(f') = 0)
            then Δ := Δ ∪ {f'}
        end;
        DF := DF \ {f}
    end
end.
```

Figure 9.2: Retracting a list of facts

Main algorithms

In this section, we present the main algorithms used in the datalog evaluation. Let `list_ded(f)` be the list of all deductions the fact f contributed to and let `Fact(d)` be the fact implied by the deduction `d`. The algorithm of Figure 9.2 computes the new fixpoint from a given Δ^-. Moreover, we denote the number of deductions implying the fact `f` by `Nb_ded(f)`. We denote the set

$$\{(r,\ i) \dagger r \in \mathrm{P}, i \geq 1, r \equiv h \text{ :- } a_1,\ \ldots,\ a_n,\ a_i \text{ is an instance of } f\}$$

by `rule_match`(f). The algorithm in Figure 9.3 computes the new fixpoint.

```
Generate_ded(Δ⁺, SD, DF, BF, SR)
begin
    Δ := Δ⁺;
    while (Δ ≠ ∅) do
    begin
        Δ := Δ \ {f};
        FB := FB ∪ {f};
        Γ := rule_match(f);
        while (Γ ≠ ∅) do
        begin
            Γ := Γ \ {(r, i))};
            Gen_ded_fact(r, i, f, Δ)
        end
    end
end.
```

Figure 9.3: Adding a list of facts

Given that the fact f matches the i^{th} atom of rule r, the subprogram `Gen_ded_fact(r, i, f, Δ)` in Figure 9.4, adds to Δ all facts that can be derived using all facts in the fact base. While executing this subprogram, the facts in $\Delta \setminus \{\mathtt{f}\}$ are not available for rule applications. The algorithm generates every possible derivation from all facts matching a given atom in r. Let us suppose that we already found a possibly empty list of facts $f_1,\ \ldots,\ f_{j-1}$ matching the atoms a_1, ..., a_{j-1} of rule r and using the substitution α. The purpose of the algorithm in Figure 9.5 is to generate all deductions of the form $r\alpha\sigma$ such that

$$\sigma = \mathrm{mgu}\{\mathrm{a}_j = \mathrm{f}_j,\ \ldots,\ \mathrm{a}_n = \mathrm{f}_n\}.$$

In addition, at termination of the algorithm, we must have $h\alpha\sigma \in \Delta$, $h\alpha\sigma \in \mathrm{DF}$ and $r\alpha\sigma \in \mathrm{SD}$.

In the algorithm of Figure 9.5, we distinguish three cases:

j = n+1 This means that we have completed a deduction. Consequently, we add it to the set of deductions SD. If the derived fact $h\alpha$ is in the fact base (FD $\cup$ BF), we simply increment its reference counter; otherwise, we add it to the fact base and to Δ.

j = i Since a_i matches with f, we recursively call `gen_case` after incrementing j.

else We try to match a_j successively with all facts in the fact base. For each matching fact, we recursively call `gen_case` after applying the corresponding substitution on the current instance of the rule.

```
Gen_ded_fact(r, i, f, Δ)
begin
    Let r ≡ h -:  a1, ..., an;
    Let σ = mgu(ai, f);
    Gen_case(r, σ, i, 0, Δ)
end.
```

Figure 9.4: Generation of all deductions from a given rule and a fact

9.4.5 Interfacing RUSSEL with the Configuration Analysis Component

In this section, we describe how RUSSEL is integrated with the configuration analysis component presented in the previous section. On the one hand, RUSSEL is augmented with an interface to the deductive component in order to enable RUSSEL rules to access or modify the current fact base; this interface is described later on. A rule can look up the fact base for a specified fact and can request the addition/removal of a specified fact in the current fact base. However, the facts that are requested for addition/removal, henceforth referred to as *scheduled facts*, are not added/removed immediately, but are collected into the two sets Δ^+ and Δ^-, respectively. Once all the rule instances that are active for the current record have been executed against the current record, the fact base is (incrementally) updated using Δ^+ and Δ^- as described in the previous section. On the other hand, the datalog language is augmented with an interface to RUSSEL to enable the configuration analysis component to trigger/cancel specified RUSSEL rules, based on the facts that have been *newly* added to or removed from the fact base. The integrated system processes the current record in three steps:

Step 1: Executing current active rules
The current record is processed by the set of active rule instances as described in Section 5.7. Let Δ^+ (resp. Δ^-) be the set of basic facts that are scheduled for addition to (resp. deletion from) the current fact base when all active rule instances have been executed against the current record.

Step 2: Updating the current fact base
The current fact base is (incrementally) updated using Δ^+ and Δ^- as described in the previous section. Let B^+ (resp. B^-) be the set of facts that are *newly* added to (resp. removed from) the fact base as a result of updating the fact base.

Step 3: Updating current rule instances
Based on B^+ and B^- obtained in step 2, certain rule instances are cancelled while new ones are added.

In the following, we describe the interface supported in RUSSEL for fact scheduling; it is utilized in step 1 above. Additionally, we describe the interface supported in Datalog for rule cancelation/triggering; it is used in step 3 above.

RUSSEL Interface:

The language RUSSEL is augmented with the three constructs $isfact(f)$, $assert(f)$, and $retract(f)$, which are used to support fact scheduling in step 1 above.

```
Gen_case(r, α, i, j, Δ)
begin
Let r ≡ h :- a₁, ..., aₙ;
if (j = n+1) then
    begin
        if (rα ∉ SD) then
            begin
                SD := SD ∪ {rα};
                if (hα ∈ BF ∪ DF) then begin DF := DF ∪ {hα}; Δ := Δ ∪ {hα} end
                else increment_ref(hα)
            end
    end
else if (j = i) then Gen_case(r, α, i, j+1, Δ)
else begin
        list_facts := find_all_facts(aⱼα);
        while (list_facts ≠ ∅) do
            begin
                list_facts := list_facts \ {f'};
                σ := mgu(f', aⱼα);
                if (σ ≠ fail)
                then Gen_case(r, ασ, i, j+1, Δ)
            end
    end
end.
```

Figure 9.5: Generation of all deductions from a given rule and a fact (*continued*)

$is\_fact(f)$

This condition allows a rule to lookup the fact base for the specified fact. It is evaluated to true if the current fact base contains the specified fact; it is evaluated to false otherwise.

$assert(f)$

This action schedules the fact f for later addition to the fact base. The result of executing this action is that Δ^+ becomes $\Delta^+ \cup \{f\}$.

$retract(f)$

This action schedules the fact f for later removal from the fact base. The result of executing this action is that Δ^- becomes $\Delta^- \cup \{f\}$.

Examples

1. **if** `is_fact(become('amo', 'root'))`
   `--> print('User amo is able to become root')`
   fi

2. **if** `EVENT = CREAT` **and** `PATH_NAME = '/var/spool/mail/root'`
   `--> assert(parentDir('/var/spool/mail', PATH_NAME))`
   fi

3. **if** `EVENT = UNLINK` **and** `PATH_NAME = '/var/spool/mail/root'`
   `--> retract(parentDir('/var/spool/mail', PATH_NAME))`
   fi

Datalog Interface:

The language Datalog is augmented with two constructs **on_new** and **on_dispose**, which are used to support rule triggering/cancelation in step 3 above.

Syntax

$$[\textbf{on\_new} \mid \textbf{on\_dispose}]\, f\, [\textbf{cancel} \mid \textbf{trigger}]\ R(Y_1, ..., Y_m)$$

where f is a fact, R is a RUSSEL rule name, and $Y_1, \cdots, Y_m$ are (restricted) terms in the Datalog language sense. The module used by RUSSEL must contain a rule declaration of the same name and arity as R.

Example: **on_new** `read(U, /dev/mem)` **trigger** `detectDevRead(U).`

Semantics

Let B^+ and B^- be the sets obtained in step 2 above. Consider all possible instances $f\sigma_1, \cdots, f\sigma_k$ ($k \geq 0$) in B^+ of the fact f, and all possible instances $f\alpha_1, \cdots, f\alpha_l$ ($l \geq 0$) in B^- of the fact f. Let $par_1, \cdots, par_n$ be the formal parameters appearing in the declaration of R, we consider the rule instance of R which is obtained by instantiating the arguments $Y_1, \cdots, Y_m$ with a given substitution, say σ. This rule instance is defined by

$$(R, \{(par_1, Y_1\sigma), \cdots, (par_n, Y_m\sigma)\})$$

For the sake of clarity, this rule instance is denoted by $R\sigma$. Consider the two sets of instances of R defined by

$$I_1 = \{R\sigma_1, \cdots, R\sigma_k\}$$
$$I_2 = \{R\alpha_1, \cdots, R\alpha_l\}$$

Finally, let I be the set of rules that are active for the next record as obtained in step 1 above. We are now in position to define the semantics of the constructs **on_new** and **on_dispose**. There are four combinations to consider according to the syntax of these constructs. In all four combinations, the effect is to modify the set I; in each combination, we depict the resultant set I.

on_new f **trigger** $R(Y_1, ..., Y_m)$
The resultant set of rule instances is $I \cup I_1$

on_new f **cancel** $R(Y_1, ..., Y_m)$
The resultant set of rule instances is $I \setminus I_1$

on_dispose f **trigger** $R(Y_1, ..., Y_m)$
The resultant set of rule instances is $I \cup I_2$

on_dispose f **cancel** $R(Y_1, ..., Y_m)$

The resultant set of rule instances is $I \setminus I_2$

The above described modification is performed for every line **on_new/on_dispose** appearing in the Datalog program and the next record is processed using the resulting set of active rule instances.

9.5 A Detailed Example

In this example we show how the deductive language is used to model the reasoning of an attacker searching for a security breach in the protection configuration of a Unix system. Additionally, we illustrate how the detection rules are triggered/canceled when a security breach is created/removed and we provide an example of such detection rules.

9.5.1 The continuous configuration analysis

```
(1)  become(U, root) :- replace(U, /etc/passwd).
(2)  become(U, root) :- replace(U, /etc/aliases).
(3)  become(U, root) :- replace(U, /var/spool/cron/crontabs).
(4)  become(U, root) :- read(U, /dev/mem).
(5)  become(U, root) :- read(U, /dev/kmem).
(6)  become(U, V)    :- start_up(V, F), replace(U, F).
(7)  replace(U, F)   :- write(U, F).
(8)  replace(U, F)   :- parentDir(D, F), replace(U, D).
(9)  read(U, F)      :- owner(V, F), become(U, V).
(10) read(U, F)      :- groupRead(F, G), inGroup(U, G).
(11) read(U, F)      :- worldRead(F).
(12) write(U, F)     :- owner(V, F), become(U, V).
(13) write(U, F)     :- groupWrite(F, G), inGroup(U, G).
(14) write(U, F)     :- worldWrite(F).
(15) inGroup(U, G)   :- groupMember(U, G).
(16) inGroup(U, G)   :- groupMember(V, G), become(U, V).
(17) inGroup(U, G)   :- replace(U, /etc/group).
(18) devRead :- read(U, /dev/mem).
(19) devRead :- read(U, /dev/kmem).
(20) rootHasMail :- parentDir(/var/spool/mail, /var/spool/mail/root).
```

Figure 9.6: Rules for modeling Unix

Figure 9.6 depicts an example of modeling the main elements of the Unix protection configuration system. Rules (1), (2) and (3) say that a user is able to acquire the privileges of the super-user if s/he is able to replace the files `/etc/passwd, /etc/aliases` or `/var/spool` respectively. Rules (4) and (5) say that a user is able to become the super-user id s/he is able to read directly the user or kernel memory file `dev/mem` and `/dev/kmem`. Rule (6) says that if a user is able to replace the start-up file of another user, then s/he is able to acquire the privileges of (or "become") that user. Rules (7) and (8) state that a user is able to replace a file if s/he is able to write to this file or if s/he is able to replace its parent directory. Rules (9), (10), and (11) mean that a user is able to read a file if he is the owner if the file or if s/he can be a member of a group granting read access to that file or if the file is world readable. Rules (12), (13) and (14) say that a file can be written by a user if the file is world writable or if s/he is the owner of that file or if s/he can become a

```
/* Illegal Read Access to Device Files */

(1) on_new devRead trigger exec.
(2) on_dispose devRead cancel exec.
(3) on_dispose devRead cancel illegal_read(X, Y).

/* Mail Vulnerability Exploitation */

(4) on_dispose rootHasMail trigger initMailExploit.
(5) on_new rootHasMail cancel stat.
(6) on_new rootHasMail cancel utimes(X, Y, Z).
(7) on_new rootHasMail cancel touch(X).
(8) on_new rootHasMail cancel create.
(9) on_new rootHasMail cancel chmod.
(10) on_new rootHasMail cancel exec_mail.
```

Figure 9.7: Dynamic Rule Activation

member of the group granting write access to that file. Rules (15), (16) and (17) say that a user is able to become a member of a group if s/he is a legitimate member of that group or if s/he can become a legitimate member of that group or if s/he can replace the file `/etc/group`. Rules (18) and (19) mean that the fact `devRead` is derived if there is at least one user able to read the user and kernel memory devices `/dev/mem` and `/dev/kmem`, respectively. Finally, rule (20) says that the fact `rootHasMail` is derived if the directory `/var/spool/mail` contains the file `/var/spool/mail/root` which means that *root*'s mailbox is not empty.

At start-up, the configuration analysis component builds the set of basic facts by scanning the file system and then, it uses the Datalog program to derive all possible facts from the collected basic facts. The fact base is then incrementally updated by analyzing the audit trail on-line. Consequently, the configuration analysis component achieves an on-line assessment of the configuration and thus it is useful even when used separately from ASAX. In the next section, we illustrate the collaboration between ASAX and the configuration analysis component.

9.5.2 The Collaboration of the two components

The successful completion of some of the intrusion scenario presented in Chapter 8 depends on a condition on the current state of the file system. Consider the detection of illegal read access to device files presented in Section 8.3.3. The successful completion of this attack scenario is possible in case the file `/dev/mem` or `/dev/kmem` grants read access to some users other than *root*. Accordingly, in the dynamic rule activation part of the datalog program depicted in Figure 9.7, line (1) means that the rule *exec* must be activated when the fact `devRead` is added to the fact base. Similarly, when this fact is retracted from the fact base, lines (2) and (3) mean that the rule *exec* as well as all possible instances of the rule *illegal_read* are cancelled. This is cancelation is appropriate since none of the device files `/dev/mem` and `/dev/kmem` are readable.

We apply the same reasoning to the mail vulnerability exploitation presented in Section 8.4. This exploitation is possible in case the file `/var/spool/mail/root` does not exist[1]. We base our discussion of this case on version 3 of the RUSSEL program *MailExploit* for detecting this intrusion

[1] This means that *root* has no mail.

and which is depicted in Figure 8.9 of Chapter 8. Line (4) means that the rule *initMailExploit* (which has exactly the same action part as the init action of *MailExploit*) must be activated upon retracting from the fact base the fact `rootHasMail`, meaning that the file `/var/spool/mail/root` no longer exists. Moreover, when the fact `rootHasMail` is added to the fact base, all instances of the rules listed in line (5) through (10) must be cancelled because the exploitation is no longer feasible.

Detecting Access to Critical Files

The fact base is updated using the RUSSEL program depicted in Figure 9.8. This program aims at detecting user actions altering the state of the protection configuration. The rule `criticalFileAccess` examines each record of the audit trail in sequence. If the current record involves an access to a security critical file or directory or to any of its parent directories (e.g., `/etc/passwd`, `/var/spool`, or a user start-up files such as `.cshrc`), the rule `update_fact_base` is triggered for the current record and uses the interfaces routines *assert*() and *retract*() in order to update the current fact base accordingly. For instance, if the current audit record reports the creation of the start-up file `~amo/.cshrc` which grants write access to all users, the facts `owner(amo, ~amo/.cshrc)` and `worldWrite(~amo/.cshrc)` are added (among other facts) to the fact base. If a subsequent record indicates that the permission modes of the file `~amo/.cshrc` have been changed so that write access is only granted to the members of the group `folon`, the fact `worldWrite(~amo/.cshrc)` is retracted from the fact base while the fact `groupWrite(~amo/.cshrc, folon)` is added. After analyzing the current record, the rule retriggers itself for the next record so that the rule will apply to every subsequent record in the audit trail. At the beginning of the analysis, the init action triggers the rule `criticalFileAccess` for the first record. The rule `update_fact_base` is depicted in Section B.5.

9.6 Implementation

9.6.1 Initializing the Fact Base

This operation consists of building the fact base by making a snapshot of the security critical pieces of the file system. This is similar to what is done in COPS [42] except that we represent security information in memory as a collection of basic facts. The next step is to compute all possible attacker goals from the datalog program. We make use of the *bottom-up semi-naive evaluation* algorithm ensuring that all inferences are made only once. The algorithm terminates when no new attacker goals can be inferred from the rules and the fact base. At that time, the basic facts describe the state of the configuration while the derived facts (or attacker goals) describe the current privileges an attacker may acquire by exploiting the corresponding breaches in the configuration.

9.6.2 Low Level Implementation

In order to make the incremental algorithms efficient, we need to store every fact and every deduction into separate data structures. Each fact provides a direct access to all deductions to which it contributes while every deduction provides an access to the fact resulting from the deduction. In addition, each predicate provides access to all rules it appears in the body of. Moreover, every fact contains a counter of the number of deductions from which the fact results. Unicity of representation of every fact and deduction is ensured by means of hashing techniques. These implementation

```
rule criticalFileAccess;
var error, event, mode, newmode:  string;
    owner, group:  string;
begin
   error := strToInt(HEADER_RET_ERROR);
   event := strToInt(HEADER_EVENT);
   owner := getOwnerName(ATTR_UID);
   group := getGroupName(ATTR_GID);
   mode := strToInt(ATTR_MODE);
   if error = 0 and (event = 10 or event = 39) /* chmod(2) or fchmod(2) */
   --> newmode := strToInt(ARG_VAL_0) /* the new permission mode */
   fi;
   if  error = 0          /* success   */
      and ( event = 4     /* creat(2)  */
         or event = 6     /* unlink(2) */
         or event = 10    /* chmod(2)  */
         or event = 39    /* fchmod(2) */
         or event = 42    /* rename(2) */
         or (73 <= event and event <= 83)) /* open(2) */
      and ( isPrefix(PATH_NAME, '/etc/aliases') = 1
         or isPrefix(PATH_NAME, '/etc/group') = 1
         or isPrefix(PATH_NAME, '/etc/passwd') = 1
         or isPrefix(PATH_NAME, '/var/spool/cron/crontabs') = 1
         or isPrefix(PATH_NAME, '/var/spool/mail/root') = 1
         or isPrefix(PATH_NAME, '/dev/mem') = 1
         or isPrefix(PATH_NAME, '/dev/kmem') = 1
         or isStartUp(PATH_NAME) = 1)
      -->
      trigger off for_current updt_fact_base(PATH_NAME, owner, group, event, mode, newmode)
   fi;
   trigger off for_next criticalFileAccess
end;

init_action;
   trigger off for_next criticalFileAccess.
```

Figure 9.8: Detecting Access to Critical Files

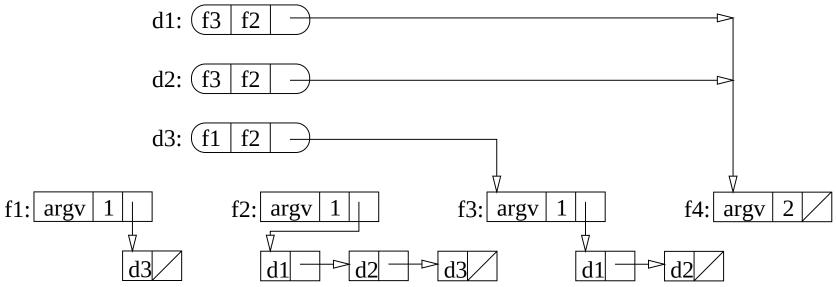

Figure 9.9: Internal Representation of the fact base

techniques are sketched in Figure 9.9, where d_1, d_2, and d_3 denote pointers to deductions and f_1 through f_4 denote pointers to facts. In this figure, f_1 and f_2 are basic facts since they are not implied by any deduction, whereas f_3 and f_4 are derived since f_3 is implied by d_3 and f_4 is implied by both d_1 and d_2. Consequently, $BF = \{f_1, f_2\}, DF = \{f_3, f_4\}$, and $SD = \{d_1, d_2, d_3\}$. Let us assume that $\Delta^+ = \emptyset$ and $\Delta^- = \{f_1\}$, we illustrate how the fact base depicted in Figure 9.9 is updated using Δ^- and Δ^+. The first step uses the algorithm shown in Figure 9.2 to compute $(\mathrm{SD}^-, \mathrm{DF}^-) = \mathrm{Ded}(BF \setminus \Delta^-, SR)$ as follows. We first remove f_1; since f_1 contributes to d_3, d_3 is removed, but since d_3 is the only deduction implying f_3, f_3 is removed as well. Now because f_3 contributes to d_1 and d_2, the removal of d_1 and d_2 decrements the deduction counter in f_4 which drops to zero and consequently f_4 is removed in turn. Since f_4 do not contribute to any deduction, the algorithm terminates. The second step in the incremental update uses the algorithm depicted in Figure 9.3, which has no effect since Δ^+ is empty. The resultant fact base is composed only of the fact f_2, which contributes to no deduction.

In order to keep track of the facts that are newly derived and the facts that are removed, every predicate has also two pointers to the list of removed and newly added facts (whether basic or derived) using that predicate. Upon termination of the fact base update, these lists represent the sets B^+ and B^-, which we described in Section 9.4.4 and used in Section 9.4.5 to modify the set of active rule instances. The process of scheduling facts for addition or removal using $assert(f)$ and $retract(f)$ is optimized as follows. When a fact is scheduled for addition, it is ignored in case it is either already in the fact base or Δ^+. Otherwise, if it is already in Δ^-, it is removed from Δ^-. If it is neither in the fact base, Δ^+, nor Δ^-, it is added to Δ^+. Similarly, when a fact is scheduled for removal, it is ignored if it is not in the fact base or if it is already in Δ^-. Otherwise, if it is in Δ^+, it is removed from Δ^+. If it is contained in the fact base and is neither in Δ^- or Δ^+, then it is added to Δ^-. As a result of this optimization, the two sets Δ^- and Δ^+ are disjoint at any time and hence, we avoid the case where the same fact is retracted from the fact base during the first step and then added to it during the second step. The sets Δ^- and Δ^+ act as caches for scheduled facts and are represented internally as two separate hash tables to speed up table lookup.

9.6.3 Accuracy of the Fact Base

To ensure the reliability of the fact base update, the fact base must correctly describe the actual state of the configuration. To ensure this, we must preselect for auditing all events susceptible to alter the state of the configuration. A challenging difficulty arises in an NFS environment wherein a user in a remote machine can mount a file system from the monitored host (which is then the

NFS server). In this case, inconsistencies may be introduced if the user modifies a security critical file or directory in the exported file system. This problem is addressed by deploying the distributed version of ASAX (see Chapter 10) on all NFS client hosts while monitoring the server. An audit record involving the modification of a security sensitive file in the mounted file system is sent to the central host where the fact base is updated accordingly. Alternatively, if the distributed auditing of all hosts is not possible, we can reinitialize the fact base periodically. Note that this also can be made incrementally.

9.7 Command Line Language

The command line language for invoking the integrated system is as described in Section 6.6.3 except that the user has now to supply the file name containing the Datalog program. The command line is as follows.

```
asax [-sa Datalog-program] [-lg] addf russel-program [nadf-file]
```

Where `Datalog-program` is the file containing the Datalog program, `addf` is the audit data description file, `russel-program` is the file containing the RUSSEL program, and `nadf-file` is the NADF file to be analyzed. Options `-s` and `-a` are exclusive. When the option `-a` is used, ASAX is invoked alone, separately from the configuration analysis component and hence, the argument `Datalog-program` must not be supplied. With option `-s Datalog-program` (the default option) ASAX is invoked in the integrated mode using `Datalog-program`; the rest of command line arguments have the same meaning and purpose as in the non integrated mode described in Section 6.6.3.

9.8 Performance Measurements

This section reports performance tests of our integrated system. The first objective of these tests is to support our belief that our approach of real-time continuous assessment of the protection configuration is feasible in a real life context. The second objective is to show that the integration of the deductive component with the intrusion detection system ASAX can reduce the total computational effort for analyzing audit trails.

The experiment was carried out on a SPARCstation-5 with 32 Megabytes of RAM running the Solaris 2.5 operating system with the Basic Security Module [51] enabled. We collected audit data for a total time period of 25 hours, 35 minutes and 42 seconds. For a period of 6 hours, 51 minutes and 48 seconds within this total time period, we used a pseudo-random generator of security-related events in order to simulate global users' activity. Every three seconds, the event generator program (executed as *root*) randomly selects either a global security-sensitive object (such as `/etc/passwd` or `/var/spool/cron`), or a user's start-up file. Then the program randomly changes permission modes of the selected file such that write access is allowed or not allowed for the members of the group, for all users, or both.

Table 9.2 summarizes the characteristics of the audit data that is used in the analysis. It shows the number of users in the configuration (Users), the number of groups (Groups), the total number of analyzed audit records (NRec), the number of audit records involving a security-related object (NSRec), the size in Megabytes of the generated audit trail (Size), the time period covered by the audit trail (Time) and the average number of audit records generated per second (NRec/sec).

Users	*Groups*	*NRec*	*NSRec*	*Size*	*Time*	*NRec/sec*
128	23	173,828	5,641	14.5	$25h : 35m : 42s$	1.89

Table 9.2: Audit Trail Description

Rules	*Exploitation*
1	Setuid program writes another setuid
2	Programs writing to executable files
3	*lpr* overwrites a file outside of /var/spool
4	Execution of known attack programs (crack, cops, etc)
5	Creation of setuid programs
6	Creation of a device file using *mknod*()
7	Writing non owned files
8	Execution of a suspicious setuid program
9	Illegal read access to /dev/kmem or /dev/mem
10	Linking an *at* job to root mail box
11	Copying a shell in root mail box when empty

Table 9.3: Summary of Detection Programs

We ran our test on the datalog program example presented in Figure 9.6 and 9.7 using all the RUSSEL detection modules presented in the Chapter 8. The purpose of these modules is remembered in Table 9.3.

Facts	*Deds*	*Size Kb*	*IFB sec*	*UFB sec*	*UPR msec*
5084	5721	568	26.52	62	0.36

Table 9.4: Fact Base Description

Table 9.4 depicts various measures related to the fact base. It shows the total number of facts (Facts) and deductions (Deds) after initializing the fact base, the total amount of memory space (Size) occupied by these facts and deductions (in Kbytes), the CPU time (IFB) for initializing the fact base from scratch (in seconds), the total CPU time (UFB) for updating the fact base during the analysis of the whole audit file (in seconds), and the average CPU time (UPR) for updating the fact base per audit record (in milliseconds). These results suggest that continuous assessment of a software configuration with our system is feasible since the time devoted to incremental deduction is negligible *wrt* the time separating the generation of two successive audit records.

Lastly, we conducted a test comparing the performance benefit of combining our ASAX intrusion detection system with the deductive component. We measured the user, system and total CPU time required by the detection programs of Table 9.3 for the analysis of the audit trail described in Table 9.2. In one case (first line in Table 9.5), the rule uses the deductive component while in the second case, it does not. These results show a one order of magnitude performance gain thanks to the fact that the detection rules are triggered only when the corresponding security vulnerabilities are introduced or in other words when the corresponding attacker goals are added to the fact base.

type	*usr sec*	*sys sec*	*total sec*	*#RPS*
Integrated	458	95.63	554.10	313.71
Not Integrated	4062.70	106.70	4169.10	41.00

Table 9.5: Integrated v.s. Non Integrated Analysis

Finally, from Table 9.5, we can see that (#RPS) the number of audit records processed per second (in the integrated case) is approximately 300 times larger that the number of records generated per second (1.18), which once again supports the feasibility of the real time assessment.

9.9 Conclusions

In this chapter, we have presented the deductive component of a programming system that aims at analyzing a computer configuration as well as its users' behaviors efficiently (in real time) and in a real life context. We use a specific —hand-made— implementation of the logic language since we believe that such a specialized implementation is needed to attain our efficiency goals. Effectiveness and practicality of the integrated system was supported by reporting performance measurements conducted on a simulated example. These measurements seem to suggest that on-line continuous analysis is feasible.

The reader may wonder why the languages respectively used for intrusion detection and for configuration analysis are so radically different. The reason is that they perform entirely different tasks: the language RUSSEL has to detect (possibly many) sequence patterns in an audit trail, in parallel, while the configuration analysis language is used for maintaining a set of logical consequences from an (evolving) set of basic facts. Thus, since efficiency is critical for real time analysis, we have chosen to design two (communicating) languages especially crafted to the tasks. Other approaches are of course possible — for instance, using a logic language for both tasks — but we prefer to follow a more pragmatic approach, which we believe easier to implement efficiently.

Chapter 10

Distributed Audit Trail Analysis

10.1 Introduction

The previous chapters demonstrated the ability of ASAX to address the single-host intrusion detection problem efficiently. They also presented a software architecture for integrating configuration analysis with intrusion detection. This chapter aims at addressing the problem of detecting intrusion patterns in a multi-host environment, wherein attack evidence may span multiple audit trails that are collected at several hosts of possibly different architectures, operating systems, and auditing facilities. Addressing the intrusion detection problem in a multi-host environment involves not only the analysis of all security-relevant events produced locally by each monitored host, but also requires intelligent correlation of related event data from multiple hosts. We present a distributed version of the language RUSSEL for supporting audit data filtering at each monitored host and for analyzing filtered audit data gathered at a central station. The presented system is conceptually universal since it does not rely on any platform feature and uses format adaptors to translate audit data streams into NADF format. The resulting system provides a powerful multi-host security monitoring system and a central interactive console for distributed analysis control, including interactive modification of the detection rule base and audit data granularity control. The hierarchical organization of the distributed system achieves a scalable architecture, which enables event abstraction at different level: users, hosts, subnets, domains, etc.

This chapter is organized as follows. Section 10.2 presents the major challenges facing auditing and intrusion detection in a multi-host heterogeneous environment. Section 10.3 discusses the functionalities that are desirable for a multi-host audit trail analysis system. The functionalities of our implemented system are described in Section 10.4. Section 10.5 presents the architecture of the distributed audit trail analysis system. Section 10.6 presents the command language of the security administrator interactive console. We outline the implementation of the distributed system in Section 10.7. Various performance measurements of the implemented system are reported in Section 10.8. Section 10.9 outlines some limitations of the current implementation and suggests possible enhancements. Finally, Section 10.10 contains the conclusion and suggests possible improvements.

10.2 Multi-host Intrusion Detection Challenges

10.2.1 Introduction

This section provides an overview of the difficulties inherent to intrusion detection in a multi-host environment. The identified difficulties serve as a basis for formulating the desirable functionalities which should be supported by a distributed intrusion detection system.

10.2.2 Heterogeneous Environment

The fact that the multi-host intrusion detection system operates in a heterogeneous environment imposes considerable constraints on the intrusion detection process, because each monitored host has its own architecture, operating system, and auditing subsystem features such as event format layout, event data semantics, audit data collection, repository, and configuration.

10.2.3 Distributed Intrusion Patterns

The intrusion patterns to be detected are determined by the set of security vulnerabilities of each monitored host. More importantly, event data supporting misuse/anomaly evidence are not necessarily contained in a single host's audit trail, but rather span several audit trails generated locally at each host. Indeed , a user behavior may be considered normal when viewed only at a single host level whereas the same behavior could reveal an outright security violation if it is viewed at the network level. In addition, host-based anomaly profiles do not scale well since user behavior thresholds may vary from one machine to another due to the fact that a user can possibly exhibit different behaviors on different hosts; a user behavior which is considered normal in a given host may be viewed as highly suspicious in another host. Moreover, a global user behavior monitoring potentially requires analyzing huge amount of audit data describing user activities that occur at each monitored host.

10.2.4 Global Time

Because of the inevitable drifts between monitored hosts' physical clocks and communication delays, the distributed intrusion detection system must cope with inconsistencies in audit data timestamps. This is of a critical importance because of the chronological nature of audit data, especially when the event patterns involve temporal properties such as temporal windows. Similarly, this lack of global time also affects temporal metrics that are used in anomaly detection such as event frequency, time series, etc.

10.3 Desirable Functionalities of a Multi-host Intrusion Detection System

This section presents high-level functionalities that should ideally be provided by a multi-host intrusion detection system.

10.3.1 Single Point Administration

Since it is not adequate to observe several analysis reports on user activities on different (virtual or physical) terminals at the same time, a global security control of the whole multi-host environment should be provided to the security administrator on a single station. Therefore, none of the security administrator's (security-related) duties should require multiple (physical or remote) sessions on several monitored hosts, he must rather be provided with a central console for interactive control of the auditing/analysis resources, detection rules, and configuration options, in a transparent way. For instance, the security administrator should be able to carry out the detection of specified intrusion patterns on selected hosts and perform various administration tasks such as auditing configuration, audit data pre-selection, and archiving, from a central station. Therefore, monitored nodes are, in a sense, considered as local objects on which administration tasks can be applied in a transparent way as if they were local to the central machine.

10.3.2 Flexible Configuration

The security monitoring should easily be customized to various environments and intrusion detection requirements. Since each monitored node has possibly different security vulnerabilities that are operating system-specific and since user behavior may vary from one host to another, the security administrator should be able to perform different kinds of analyses on different hosts. Moreover the security administrator should be able to flexibly modify the applied analyses at various levels: processes, sessions, users, hosts, the entire distributed environment. That is, during operation, the intrusion detection process should interactively be adapted to changing needs. In particular, such adaptation should not require recompiling/relinking the intrusion detection software or other related utility programs.

10.3.3 Scalability

Scalability means the ability of the intrusion detection system to modify its structure and functionalities to take into account the size and structure of the distributed environment to be monitored. For instance, scalability should allow the security administrator to perform a hierarchical evaluation of multiple hosts, thereby enabling different levels of abstractions of security-related reports. The highest level — the whole distributed environment level — of the hierarchy should represent the highest abstraction involving consolidated reports about the security status of the whole monitored environment. The lowest level — the process level — of the hierarchy should depict status reports about possible intrusive behavior of the considered process. Scalable intrusion detection in a distributed environment is especially needed because current environments tend to have dynamic hierarchical structures to accommodate changing needs.

10.3.4 Availability

In highly security-sensitive environments, security administrators need a high degree of availability of the security monitoring system in order to ensure a continuous control of security-related events. Availability means that the failure of any monitored host must have a minimal or no effect on the monitoring of other hosts. If a host were to go down for some reason, the security analysis and reporting must be able to continue on the rest of monitored hosts and the central interactive console station must not be restarted. In addition, if the central station encounters a severe failure (reboot, power outage, etc.) the security administrator should be able to recover the analysis from another available station, hence resuming the analysis from the time prior to the crash (i.e., the distributed intrusion detection will still be completely performed albeit delayed).

10.3.5 Portability

The distributed on-line analysis system should be as independent as possible of any platform feature (such as machine architecture, operating system, auditing facility, etc.), thereby providing monitoring of heterogeneous environments. Provisions should be made to allow the portage of the system on existing platforms with a low (re)programming effort.

10.3.6 Security

Security of the multi-host intrusion detection system means ensuring the integrity and confidentiality of the audit data storage and transmission as well as the integrity of the distributed monitoring system itself. Therefore, audit data and security reports must be protected against corruption and/or unauthorized access. The whole effort devoted to audit trail analysis is meaningless if appropriate access control to audit data is not enforced. Moreover, audit data transmission between monitored hosts and the central station must be protected against users listening to the transmission media and an authentication mechanism should be used in order to avoid impersonation.

10.4 Our Approach

The solution proposed in this thesis completely met requirements 10.3.1, 10.3.2, 10.3.3 and 10.3.4 (Single Point Administration, Flexible Configuration, Scalability, and Availability). Requirement 10.3.5 (Portability) is addressed at the logical level: architecture and language. The only platform-dependent parts of the system are the audit data input process and the audit data pre-selection process. However, the implementation isolates these parts into well identified modules to ensure "portability with a minimum reprogramming effort". The implementation of the distributed on-line intrusion detection system uses a distributed version of ASAX to filter audit data at each host and to analyze filtered data gathered at a central station. The prototype implementation of the system applies to a local area network of Sun workstations running the SunOS 4.1.1 operating system with its C2 security level auditing mechanism. The following sections detail the functionalities of the implemented prototype and its architecture.

10.4.1 Local Audit Data Processing

The monitoring system uses the language RUSSEL to locally analyze audit data collected at each host. This is achieved by applying RUSSEL programs on local audit files; both RUSSEL programs and audit files are security administrator-selectable. Local audit data analysis processes operate independently and have their local view which is restricted to the scope of a single machine. Local analysis is also called *filtering* since at the network-level, it serves as a pre-selection stage which relieves the central station from a large amount of processing and management tasks. However, the filtering stage may involve intelligent filtering of local audit data, therefore, the resulting filtered events may correspond to any complex event patterns rather than simple record-wise selection of events. Due to the powerful analysis capability of RUSSEL, as much complex local processing as possible should be enabled in order to achieve a better load balancing between local filtering and global analysis. The event stream resulting from a local analysis normally comprises raw audit events. However, it is possible to define more elaborate, or condensed, event types which are more appropriate vehicles for aggregate event patterns. Such condensed event types may be expressed in terms of event classes rather than basic event types, user sessions start/termination, summary data such as the accumulated number of system calls per process during a user session, etc. These condensed events are especially useful as a basis for implementing distributed historical user activity profiles, which are then consolidated upon reaching the central station. Consequently, the process of filtering local audit data not only relieves the task of the central station, but also yields higher level abstraction of security-related events.

10.4.2 Global Analysis

Audit records filtered at various nodes are transmitted to a central host where a global (network level) analysis takes place. In its most interesting use, global analysis aims at detecting patterns related to global network security status rather than host security status. In this regard, global analysis encompasses a higher level and a more elaborate notion of security event.

Concerted local and global analysis approach lends itself naturally to a hierarchical model of security events in which components of a pattern are detected at a lower level and a more aggregate pattern is derived at the second higher level and so on. Note that an aggregate pattern could exhibit a malicious security event while corresponding sub-patterns do not at all. For instance, a login failure by a user is not an outright security violation but the fact that this same user is trying to connect to an abnormally high number of hosts may indicate that a network attack is under course. Organizations often use networks of interconnected Lans corresponding to departments. The hierarchical model can be mapped on the organization hierarchy by applying a distributed analysis on each of the Lans and an organization-wide analysis carried out on audit data filtered at each Lan. For a large number of nodes, intermediate audit data processing stations which form a tree-like structure might be inserted. Thus, concerted filtering and global analysis can lead to the detection of very complex patterns. In addition globally analyzing audit data, the central station also handles control messages for local analyses initiation and termination as well as other configuration control messages.

In the following, the node performing the global analysis is called the *central* or *master* machine while filtering takes place at *slave* machines. Correspondingly, we will also refer to master and slave evaluators. A *distributed evaluator* is a master evaluator together with its associated slave evaluators.

10.4.3 Audit Data Granularity Control

During operation of the multi-host audit trail analysis system, it is possible to alter audit data granularity at different levels: processes, user sessions, hosts, and the whole environment. For instance, upon detecting suspicious activities, the security administrator should have the ability to increase the level of detail of audit data collection, either interactively from the central station or dynamically through intrusion detection rules.

10.5 Architecture

The architecture of the distributed system is addressed at two different levels. At the host level, a number of processes cooperate to achieve logging control and filtering. The global architecture supports the network level analysis. This section aims at giving an intuitive view of the overall distributed system. Both global and local analyses are performed by a slightly modified version of the analysis tool outlined in the previous section.

10.5.1 Host level

Processes in the local architecture are involved in the generation of audit data, control of its granularity level, conversion of audit data to NADF format, analysis of audit records and finally transmission of filtered sequences to the central evaluator.

Audit trail generation

This mechanism is operating system dependent. It generates audit records representing events such as operations on files, administrative actions, etc. It is assumed that all monitored hosts provide auditing capabilities and mechanism for controlling granularity level. The process generating audit records is called the *audit daemon* (*auditd* for short).

Login controller

This process communicates with *auditd* in order to alter the granularity. It is able to change the set of pre-selected events. This can be done on a user, host and network basis. Furthermore, we distinguish between a temporary change which applies to the current login session and a permanent change affecting also all subsequent sessions.

Format adaptor

This process translates audit trails generated by *auditd* to the NADF format. Native files can be erased after being converted since they are semantically redundant with NADF files. Keeping converted files instead of native files has several advantages: the files are converted only once and can be reanalyzed several times without requiring a new conversion. Moreover, in the context of an heterogeneous network, they provide a standard and unique format. However, NADF files are normally greater[1] in size than native files.

Local evaluator

It analyzes the NADF files generated by the format adaptor. Note that several instances of the evaluator can be active at the same time to perform analyses on different NADF files or possibly on the same file. Off-line and on-line analyses are implemented in the same way. The only difference is that in on-line mode, the evaluator analyzes the currently generated file. These processes will be further described in section 10.7.
Audit records filtered by slave evaluators on the various monitored slave machines are sent to the central machine for global analysis.

10.5.2 Network level

At the network level, the system consists of one or more slave machines running the processes previously described and a master machine running the master evaluator (see Figure 10.1).

The latter performs global analysis on the audit record stream resulting from local filtering. The result of the central analysis can be network security status reports, alarms and statistics, etc. In addition, a console process is run on the master machine. It provides an interactive command

[1]This is due to the additional storage space required for the added audit data *ids*, audit data *length* fields, and the total length of the NADF record. However, one may be interested only in a subset of event data fields contained in audit records and this reduces the size of the NADF file.

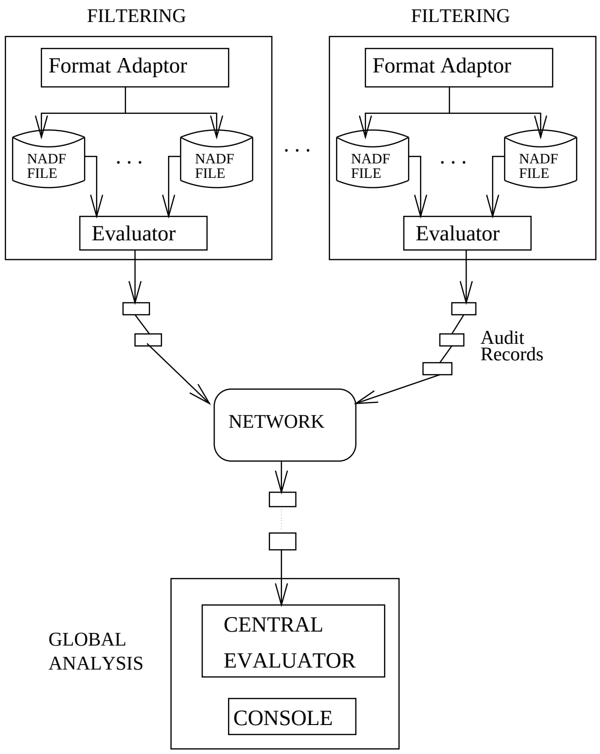

Figure 10.1: System Architecture

interface to the distributed monitoring system. This command interface is briefly described in the next section.

10.6 The Command Language of the Distributed System

10.6.1 Preliminaries

This section presents the command interface used by the security officer. In the following, evaluator instances are identified by their instance numbers which are similar to Process IDs in UNIX systems. Auditable events are determined by a comma separated list of audit flags which are borrowed from the SunOS 4.1 C2 security notation for event classes. (The SunOS 4.1 C2 security features are described in detail in [52].) These audit flags are listed in Table 10.1.

Audit flags can optionally be preceded by + (resp. -) to select only successful (resp. failed) events. For instance, the list of audit flags *+dr,-dw,lo,p0,p1* specifies that successful data read, failed data writes, all new logins and all privileged operations are selected. Under SunOS, the file `/etc/security/audit/audit_control` contains (among other things) a list of audit flags determining the set of auditable events for all users of the system. `/etc/security/passwd.adjunct` is called the *shadow* file and contains a line per user which indicates events to be audited for this particular user. The actual set of auditable events for a given user is derived from the system audit value and the user audit value according to some priority rules. Finally, audit trails in NADF format respect naming conventions based on creation and closing times. This allows to easily se-

flags	short description	example
dr	data read	*stat*(2)
dw	data write	*utimes*(2)
dc	object create/delete	*mkdir*(2)
da	object access change	*chmod*(2)
lo	Login, Logout	*login*(1)
ad	Administrative operation	*su*(1)
p0	Privileged operation	*quota*(1)
p1	Unusual operation	*reboot*(2)

Table 10.1: SunOS C2 security audit flags

lect files generated during a given time interval. For instance, the file named $time_i.time_f.NADF$ contains events generated by *auditd* in the time interval [$time_i$, $time_f$]. Supported commands fall into two categories:

10.6.2 Analysis control commands

The commands for distributed analysis allow the security administrator to start, stop and modify a distributed analysis interactively. A distributed analysis is completely determined by the set of slave hosts, the master host (which is normally the host on which the interactive console is running), the RUSSEL programs that are to be executed on each of these hosts, and finally a time interval where the audit data must be contained. To start a new distributed analysis on a set of monitored hosts, one first prepares a text file specifying the involved hosts, the RUSSEL modules to be applied on the hosts, and optionally an auditing period (a time interval which is the same for each node). By default, the analysis is performed on-line. This file is given as an argument to the **run** command.

Using the **rerun** command, the security officer can change attributes of an active distributed evaluator either by changing rule modules on some hosts (master or slave) or by changing the time interval used by the whole distributed evaluator. The **rerun** command is parameterized by an evaluator instance number and a rule module or a time interval. It is possible to activate several distributed evaluators which run concurrently and independently of each other.

The **kill** command stops an evaluator identified by its instance number. To eliminate orphan filters, killing a master evaluator automatically cleans up all its slave evaluators. The **ps** commands reports the attributes of all currently active distributed evaluators. Attributes of an evaluator include instance number, instance number of the corresponding master evaluator, host name, rule module and time interval. The command **reset** stops all current distributed evaluators.

10.6.3 Logging control commands

The command **logcntl** implements the granularity control functionality (see 10.7.2). It allows the security administrator to alter the granularity level for any monitored user or host. The security officer is so able to change the auditable events for a particular user on a particular host according to the list of audit flags supplied to **logcntl**. With the option *-t* the change takes effect immediately, however, the settings are in effect only during the current login session. With the option *-p*, the change takes effect the next time the user logs in and for every subsequent login session until this command is invoked again. On the host basis, the security officer can alter the system audit value of a specified host by supplying a host name and a list of audit flags.

Although the **logcntl** command relies closely on the SunOS formalism for specifying auditable events and altering the set of events currently audited, it could be possible to develop a system-independent event classification as well as a portable auditing configuration. Nevertheless, as the SunOS 4.1 uses an event classification and auditing configuration that are similar to most O.S, the current solution is sufficient for a prototype system.

10.6.4 Example

In this section, the failed login detection example introduced in Chapter 3 (Informal Semantics of RUSSEL) is reconsidered in the context of a distributed analysis. The purpose is still to detect repeated failed login attempts, but now failed login events can occur at any of the monitored hosts (here we consider two hosts viz. *poireau* and *epinard*). According to the filtering/global analysis principle, a slave evaluator is activated on each hosts (*poireau* and *epinard*) and a master evaluator is initiated on *poireau*. Each slave evaluator only filters failed login records from its local host and sends it to the master evaluator which then analyzes the filtered record stream to detect the sequence of failed logins. As indicated in the evaluator description file shown in Figure 10.2, filtering is implemented in RUSSEL by the rule module *badlogin.asa* while the sequence of failed logins is detected using the rule module *nbbadlogin.asa*. This file also contains the time interval to which

```
master poireau:  nbbadlogin:[19940531170431,19940601173829];
slaves poireau, epinard:  badlogin.
```

Figure 10.2: Distributed Analysis Description File

analysis is applied. Figures 10.3 and 10.4 depict the content of *badlogin.asa* and *nbbadlogin.asa* respectively.

Notice that the master evaluator does not check that records correspond to login failure events since this is already done by the associated slave evaluators. Figure 10.5 shows how the distributed evaluator is activated using the interactive console window. The lower window contains the distributed analysis interactive console. The security officer has just invoked the **run** command with the name of the evaluator description file as argument. The upper window is the Unix console where outputs from the master evaluator are printed.

10.7 Overview of the Implementation

The implementation of inter-process communication in the distributed system distinguishes between two types of processes, based on their tasks.

- *audit data analysis processes*: audit data retriever, or supplier, processes, (slave and master) evaluator processes, and the console process.
- *audit data management processes*: format adaptors, audit data granularity controllers, and audit trail size controllers.

On the one hand, these two types of processes require different privileges: audit data analysis processes require the privileges of user `audit` for accessing audit data analysis programs and audit

```
rule failed_login;
begin
  if         EVENT = 'login_logout'
    and AU_TEXT_4 = 'incorrect password'
    --> send_current
  fi;
  trigger off for_next failed_login
end;

init_action;
begin
  trigger off for_next failed_login
end.
```

Figure 10.3: Slave evaluator module: badlogin.asa

files, whereas for instance, audit data granularity control requires root privileges for modifying user event pre-selection masks. On the other hand, audit data management processes must always be active to ensure for instance that all audit records are converted to NADF format whereas audit data analysis processes may be activated and stopped by the security administrator. The implementation of inter-process communication between analysis processes uses PVM (Parallel Virtual Machine) [53], a library of network communication routines supporting initiation/termination of processes as well as communication and synchronization between processes. In PVM, inter-process communication is done by message passing. In order to be a member of the Parallel Virtual Machine, a process must enroll itself in the Parallel Virtual Machine by calling a PVM library routine. Multiple instances of the same executable file are distinguished by their unique instance numbers. PVM attaches a *receive* buffer and a *send* buffer for each process participating in the Parallel Virtual Machine. The receive buffer is used to store incoming messages and the send buffer contains messages to be sent to other processes. The reader is referred to [53] for further details.

The use of PVM is justified because it provides a low development cost compared with low-level inter-process communication interfaces such as sockets. A notable limitation of PVM is that all processes must belong to the same user and the processes initiated by two distinct users cannot communicate with each other. For this reason and because certain audit data management processes must run under *root*'s identity, we use RPC for implementing the communication between these processes and the console process.

In the following, the steps for analyzing and filtering user activities starting from local processing, transferring the results, global audit data processing, and the display of results at the central station are detailed. In addition to the *auditd* process, the following concurrent processes are attached to each monitored host (see Figure 10.6):

10.7.1 Distributed format adaptor (FA)

The distributed format adaptor *fadapter* translates SunOS audit files into NADF format. It also observes date and time-based naming conventions for NADF files: a NADF file consisting of the chronological sequence of audit records R_0, ..., R_{n-1} has a name of the form

$$time_0.time_n.host.NADF$$

```
global v: integer;

rule failed_login(max_times, duration: integer);
begin
  trigger off for_next count_rule(AU_TEXT_3, strToInt(AU_TIME)+duration, max_times-1)
end;
rule count_rule(target_uid: string; expiration, count_down: integer);

if              AU_TEXT_3 = target_uid
  and strToInt(AU_TIME) < expiration
  --> if count_down > 1
        --> trigger off for_next count_rule(target_uid, expiration, count_down-1);
        count_down = 1
        --> begin
              v := v + 1;
              println(gettime(AU_TIME), ':  3 FAILED LOGINS ON ', target_uid);
              trigger off for_next failed_login(3,120)
            end
      fi;
  strToInt(AU_TIME) > expiration
  --> trigger off for_next failed_login(3,120);
  true
  --> trigger off for_next count_rule(target_uid, expiration, count_down)
fi;

rule print_results;
begin
  println(v, ' sequence(s) of bad logins found')
end;

init_action;
begin
  v := 0;
  trigger off for_next failed_login(3, 120);
  trigger off at_completion print_results
end.
```

Figure 10.4: Master evaluator module: nbbadlogin.asa

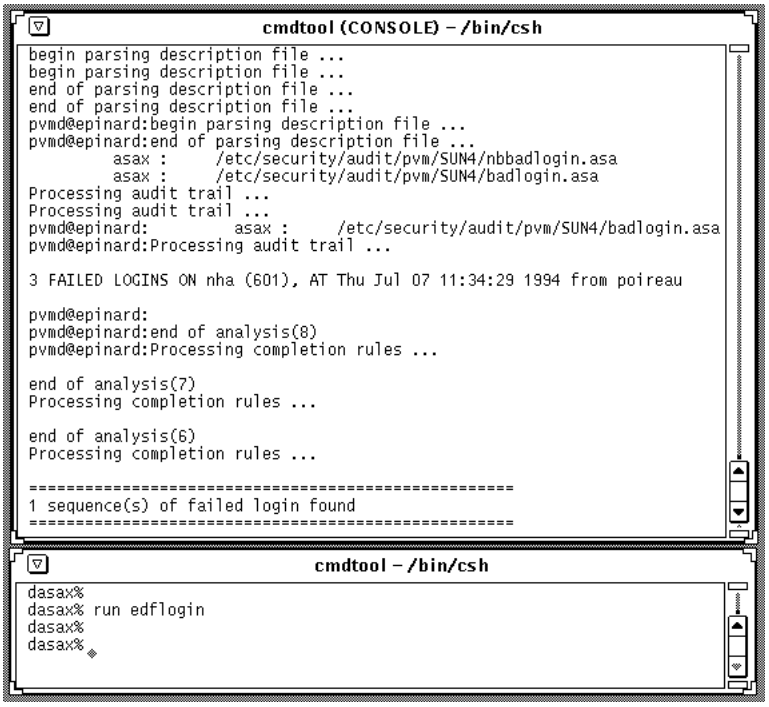

Figure 10.5: Console windows

where $time_0$ is the time stamp found in R_0, $time_n$ is the time stamp in R_{n-1} plus one second, and *host* is the host name where the format adaptor is running. Both $time_0$ and $time_n$ are 4 decimal digits year, and 2 decimal digits for each of the month, day, minute, and second. The current NADF file has a name of the form

$$time_0.not\_terminated.host.NADF$$

where $time_0$ is the time stamp of its first record.
The current native and NADF files are limited to a maximum size which is recorded in the file *nadf_data*. The process *sizer* sends a signal to *auditd* (resp. *fadapter*) if the maximum size for the current native (resp. NADF) file is reached. When *auditd* or *fadapter* receives such a signal, it closes the current file and continues on a new one. The maximum size can be changed at any time by a simple RPC (*Remote Procedure Call*) server *dflsize_svc* after request from the console process. *dflsize_svc* updates the file *nadf_data* accordingly.
The distributed FA is automatically started at boot time of each monitored host from `/etc/rc.local`.

10.7.2 Logging control

Changing the granularity level for a user or a host is performed remotely from the security officer console by a remote update of the *auditd* configuration of the involved host. Therefore, logging control is implemented by means of RPC. For this purpose, to each monitored host is attached a server process *logcntl_svc* accepting requests from the console process running on the master machine. Depending on the option used for the command **logcntl**, the console process calls an

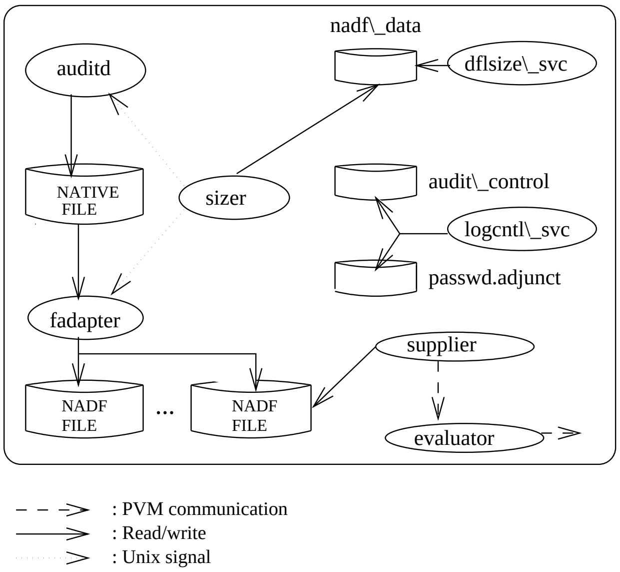

Figure 10.6: Local Architecture

appropriate procedure offered by the *logcntl_svc* server on the involved host. According to the RPC model, *logcntl_svc* transfers control to the appropriate service procedure and then sends back a reply to the console process indicating the outcome of the call.

It was not possible to implement such a communication using PVM since all processes participating in the Parallel Virtual Machine must belong to the same user while the *logcntl_svc* server requires root privileges to access the shadow password file. Moreover, the security officer should not necessarily own root privileges.

10.7.3 Supplier process

This process is started by the console process on each monitored host and is responsible for supplying audit records contained in the local audit data repository to the evaluator, called its *peer* process, running locally on the same machine. The console process transmits to each supplier process the instance number of its peer evaluator. The console process then sends a time interval to each supplier process. The latter retrieves from local NADF files the audit records corresponding to this time interval and then sends them to its peer evaluator; each audit record being sent in a separate PVM message. It is interesting to note that slave and master evaluators are implemented exactly by the same code. This is possible at the cost of providing the additional supplier process which hides the details of how audit records are retrieved. For slave evaluators, the records are received from the supplier process while a master evaluator receives them from its slave evaluators.

10.7.4 Evaluator process

The evaluator process (on master and slave machines) is the heart of the distributed system. It analyzes record streams according to a rule module. The master evaluator analyzes a record stream originating from a set of slave evaluators and the result of the analysis may be reports, alarms, statistics, etc. In order to achieve an improved interlacing of the audit records received by the master evaluator, the latter reads one audit record at a time from each slave evaluator in a round-robin fashion. A slave evaluator analyzes a record stream originating only from its peer supplier process and the result of the analysis is a filtered sequence of audit records which are sent to the master evaluator. The console process can change the rule module used by an evaluator by sending to it the name of the new module to be applied. At reception, the evaluator executes the completion rules, compiles the new module, executes the resulting init-actions and then waits for audit records. The time interval can also be changed for all evaluators participating in a distributed analysis. For this purpose, the console process sends the new time interval to all involved supplier processes and notifies evaluators for such a change. Upon reception, supplier processes send to their respective peer evaluators the record stream determined by the new time interval. Evaluator processes only execute completion rules and init-actions. Completion rules report the results of the previous analysis before changing the current rule module or the time interval.

10.7.5 Console process

This process was already partially described in previous sections. Only a single instance of the console process exists and is active on the master machine under control of the security officer through the command interface described in section 10.6. It maintains the status of all active distributed evaluators and coordinates all processes of the distributed system. Under interactive control of the security officer, the console process can also invoke the remote *logcntl_svc* RPC server to change the current granularity level on a given host.

To activate a distributed analysis as indicated in a distributed evaluator description file, the console process initiates an evaluator-supplier pair on each slave host and a master evaluator on the master host. It then sends the time interval to all supplier instances and the appropriate rule module to each evaluator instance. When all suppliers are positioned in the time interval and evaluators have successfully compiled their modules, the console process starts the analysis by triggering record stream transmissions from suppliers to slave evaluators.

10.8 Performance Measurements

10.8.1 Introduction

This section reports some performance tests of our system. These measurements aim at showing the feasibility and effectiveness of the distributed system in terms of response time and network load. It will also follow from these measurements that on-line monitoring is feasible.
The experiments were carried out on two SUN SPARCstation 1 running the C2 security level of the SunOS 4.1 and connected to a 10 Mbytes/sec Ethernet. Each machine has 16 Mbytes of random access memory. In addition, a third machine on the Ethernet is used as a file server where NADF files generated at each host are stored using NFS (*Network File System*).

The first experiment measures the overhead introduced by transmitting audit records from slave evaluators to the master evaluator using PVM in comparison with the case where the same analysis is performed using the single host version of ASAX. The second experiment compares the performance of a distributed audit trail analysis with that of a centralized audit trail analysis. In the centralized analysis, monitored nodes do not perform any local processing of audit data; they send the whole local audit records falling into a given time interval to the central station without filtering. The purpose of this experiment is to highlight the performance gain resulting from concerted local and global analyses and also shows that the distributed analysis reduces the network traffic significantly. The third experiment shows the performance gain resulting from several analyses executed in parallel compared with the case where these analyses are executed in sequence one after the other.

10.8.2 Overhead of the Distributed Architecture

The experiment described in [54] compares the communication times for a number of different network programming environments on isolated two and four nodes networks. Since messages exchanged in the distributed system are around 300 bytes in size, it follows from the measurements conducted in [54] that the average data transfer rate is around 0.049Mbytes/sec. In order to measure the network communication overhead introduced by the distributed architecture, we analyzed a single audit file using the single host version of ASAX on the one hand and the distributed version on the other hand. The purpose of both analyses is to detect three successive failed login attempts within a time period of three minutes and in both cases we analyzed a 500 Kbytes audit file corresponding to two days usage of the system by two users. Audited events are file operations as well as normal administrative operations such as the *su* and *login* commands. The single host analysis uses the RUSSEL program shown in Figure 4.1. The distributed analysis uses a single slave evaluator and a master evaluator. The slave applies the *badlogin.asa* module shown in Figure 10.3 and the master applies the *badlogin.asa* module shown in Figure 10.4. The overhead introduced is mainly due to network communication (using PVM) between the slave and the master. Table 10.2 gives the mean values of the CPU and elapsed times (in seconds) for the single host analysis (SHA) and the distributed analysis (DA).

The results suggest that the distributed audit trail analysis is feasible since the elapsed time for the analysis is negligible *wrt* the time spent in generating the audit data (2 days). However, the overhead due to the distributed architecture is significant: most of the elapsed time is spent in communication between nodes. Consequently, improvements of the distributed system response time involves optimization of the network communication.

type	*usr*	*sys*	*total*	*elapsed*
SHA	1.13	0.68	1.81	5.3
DA	3.43	3.73	7.20	55.7

Table 10.2: Single Host Analysis v.s Distributed Analysis

10.8.3 Centralized v.s distributed audit trail analysis

This section reports the performance benefits of a distributed network security monitoring over a centralized network security monitoring. In the latter approach, monitored nodes do not perform any intelligent[2] filtering of audit data. All audit records generated at one node are sent to a central host where the analysis takes place. As shown in Table 10.3, the distributed analysis has the advantage of drastically reducing the network traffic in comparison with the centralized analysis (CA). It also achieves a balancing of the CPU time over several machines. The CPU time of the master evaluator is smaller since part of the analysis is carried out by slave evaluators on slave

[2]Assuming that a simple pre-selection of auditable events cannot be considered as an intelligent filtering.

machines. A system using a centralized architecture for network audit trail analysis is presented in [55].

type	*usr*	*sys*	*total*	*elapsed*	*traffic*[a]
CA	11.90	13.60	25.56	265.78	2,661
DA	1.15	7.46	8.61	188.56	39

[a]In Kbytes

Table 10.3: Distributed v.s Centralized Analysis

10.8.4 Parallel v.s sequential analysis

The language RUSSEL allows us to execute more than one analysis at the same time i.e., during a single analysis of a given audit file, several independent rule modules can be executed. For instance, we can search in parallel for repeated failed logins as well as for repeated attempts to corrupt system files.
We used 4 distributed evaluators described by their distributed evaluator description files. All analyses are limited to a specified time interval as shown in Figure 10.7.

```
bad su commands:
master poireau:  nbbadsu:[19940531170524,19940606173854];
slaves poireau, epinard:  badsu.

System corruption:

master poireau:  fscorrupt:[19940531170524,19940606173854];
slaves poireau, epinard:  corrupt.

Set user id files:

master poireau:  setuid:[19940531170524,19940606173854];
slaves poireau, epinard:  create.

trojan su:

master poireau:  trojan:[19940531170524,19940606173854];
slaves poireau, epinard:  exec.
```

Figure 10.7: Multiple Distributed Analyses: RUSSEL modules for the master and the slaves

The purpose of the first one is to detect 3 repeated failures to break a given account using the *su* command. Each of the hosts *poireau* and *epinard* runs a slave evaluator which detect unsuccessful *su* commands. The master evaluator detects sequences of 3 failed *su* commands invoked at any of the two monitored hosts.

The purpose of the second distributed analysis is to detect attempts to corrupt system files on either of the two hosts. System files corruption could be deletion, creation, attribute modification of any of system files or directories. Each slave evaluator applies the RUSSEL module *corrupt.asa* that detects deletion, creation or attribute modification of files. The master evaluators uses the module *fscorrupt.asa* to check that such operations involve a system file.

The third analysis aims at detecting new set user id files. For this purpose, slave evaluators on *epinard* and *poireau* detect creation of files on hidden directories such as */tmp* or */usr/tmp* and modification of their access flags. At the master evaluator, the module *setuid.asa* is used to detect the creation of a file on hidden directory followed by a modification of access flags of these same file such that the created file is a set user id file.
The last analysis searches for trojan system programs such as *su*. The slaves detect the execution of any command using the module *exec.asa* while the master applies the module *trojan.asa* to check if such an execution involves a trojan program.

The multiple distributed analysis amounts to execute these distributed analyses one after the other using the **run** command with the appropriate distributed evaluator description file as argument. The corresponding execution times are reported in Table 10.4. In the case of parallel execution, we activate a single distributed evaluator which performs the four analyses at the same time. For this purpose, the master evaluator uses a rule module (*master_module.asa*) which includes all modules applied by each of the above 4 masters (see Figure 10.8).

```
master_module.asa:
uses nbbadsu, fscorrupt, setuid, trojan.

slave_module.asa:
uses badsu, corrupt, create, exec.
```

Figure 10.8: Parallel analysis: RUSSEL modules for the master and slave

type	*usr*	*sys*	*total*	*elapsed*
nbbadsu	02.43	10.18	12.61	159.48
fscorrupt	02.33	12.51	14.48	176.90
setuid	03.10	11.98	15.08	182.46
trojan	02.83	11.83	14.66	184.09
total	10.69	46.50	57.19	702.92
parallel	08.03	15.03	23.06	209.83

Table 10.4: Multiple Distributed Analysis v.s Parallel Analysis

Similarly, slave evaluators (on *epinard* and *poireau*) run a single module (*slave_module.asa*) which includes the 4 ones applied by the previous slave evaluators. The distributed evaluator description file for the parallel execution is depicted in Figure 10.9.

The execution times for the parallel analysis are found in the last line of table 10.4. It follows from this table that the performance gain is substantial. Note that the elapsed time of the parallel

```
parallel distributed evaluator:
master poireau:  master_module:[19940531170524,19940606173854];
slaves poireau, epinard:  slave_module.
```

Figure 10.9: Parallel Analysis Description File

analysis is not significantly different from the elapsed time of a single analysis. This suggests that complex on-line analyses (combining many single analyses in parallel) are feasible.

10.9 Limitations of our Approach

The approach proposed in this chapter attained the core functionalities described in Section 10.4; other functionalities have not been addressed which we discuss in the following sections.

10.9.1 Clock Synchronization

We have not addressed the lack of global time (see 10.2.4) in the present implementation of the system. Future versions of the system will make use of common distributed algorithms for achieving clock synchronization between all hosts participating in the monitoring system (master and slaves). For instance, TEMPO [56] uses a master-slave architecture of a distributed service that synchronizes the clocks of Berkeley UNIX 4.3BSD systems. The accuracy of the clock synchronization algorithm is a function of the network transmission delay and depends linearly upon the drift rate of the clocks and the interval between synchronizations rounds. Based on a sufficiently accurate clock synchronization[3], we can address the problem of achieving a merged event stream at the master evaluator which is as chronologically ordered as possible. In the solution we plan to use, the master evaluator broadcasts fixed but selectable time intervals to all slaves. Upon receipt of the time interval, a slave evaluator transmits only events falling within the time interval. Additionally, after expiration of the time interval, the master evaluator then can sort the received chunk of event stream to ensure total order in time stamps. Alternatively, the master evaluator can process the received chunk as it is without reordering it. The former solution is more reliable but is expected to introduce a non negligible overhead whereas the latter may be an acceptable approximation of the total order, especially if the length of the time interval is relatively small.

10.9.2 Security and Integrity Issues

The security requirement of the distributed audit trail analysis system (see 10.3.6) was only partially addressed using common UNIX file system protection mechanism whereas the integrity and confidentiality of transmitted audit data between slaves and master evaluators were not addressed at all. Future version of the system will use the SSL (Secure Socket Layer) for achieving the desired security goals.

[3]In [56], the authors reported that if TEMPO synchronizes the clocks every 4 minutes, the maximum error of the clocks is 30 milliseconds.

10.9.3 Distribution of Audit Data Analyses

In this chapter we did not provide specific guidelines for adequate distribution of the analysis with regard to achieving an optimal load balancing between slave evaluators and the master evaluator. An optimal load balancing consists of evenly distributing the analysis computation overhead between slaves and master evaluators so that the overall response time of the monitoring system is at a maximum. An empirical tuning of the system, based on experimentations, could be utilized as a first step in approaching this problem. A distributed RUSSEL program design methodology constitutes an interesting future research direction towards a more general solution to this problem.

10.10 Conclusions and Future Works

This chapter presented an implemented system for on-line analysis of multiple distributed data streams. Universality of the system makes it conceptually independent from any architecture or operating system. This is achieved by means of format adaptors which translate data streams to a canonical format. The rule-based language (RUSSEL) which is specifically designed for analyzing unstructured data streams makes the presented system (theoretically) as powerful as possible and still efficient enough for solving complex queries on the data streams. We also presented the distributed architecture of the system and its implementation.

Effectiveness of the distributed system was demonstrated by reporting performance measurements conducted on real network attack examples. These measurements also showed that on-line distributed analysis is feasible even for complex problems. Further works will tackle the problem of reducing the overhead due to network communication. For the present version of the system, audit records are transmitted using one PVM message by record. A first improvement is to buffer audit records before packing them in a single PVM message. Another improvement involves a direct use of standard communication protocols such as TCP/IP instead of PVM. More standard protocols will increase the portability and the robustness of our system. We also presented some limitations of the current version of the system and also discussed possible perspectives for overcoming these limitations in future versions of the system.

Chapter 11

A Format Adaptor Generator

11.1 Introduction

To implement a format adaptor, a programmer has to invest a significant effort writing a translator of an audit file from the native format into the NADF format. The programmer must also provide the audit data description file mapping external field names to internal representation. However, the translation is usually tedious, time consuming, and error prone. It would be very convenient if a software tool could automatically perform the task of generating the source code of the format adaptor. Motivated by this idea, a format adaptor generator, named *fagen* , has been designed. *fagen* compiles a native format description and generates the C-source code of the format adaptor corresponding to the given description. In addition, *fagen* produces the audit data description file corresponding to the NADF file. *fagen* uses a format description language (FDL), a home-grown language for convenient and powerful description of native audit files. Therefore, using *fagen* has the benefit of substantially reducing the cost of implementing format adaptors.

In Section 11.2, we explain the motivations and requirements for *fagen.* In Section 11.3, we describe the design choices. In Section 11.4, we provide an informal description of the format description language FDL and *fagen.* Section 11.5 contains the syntax of FDL and provides some examples illustrating its use. Sections 11.6, 11.7, and 11.8, present the translation rules for generating the format adaptor and the audit data description file. Section 11.9 outlines the process of C-source code generation from an internal description of the native format. Finally, Section 11.10 discusses some limitations of *fagen* and draws conclusions.

11.2 Motivations and Requirements

The implementation of a format adaptor may be tedious because an auditing mechanism supports numerous audit record types each of which has a different layout and structure. (For instance, BSM 2.5 [41] supports 280 event types.) Therefore, a format adaptor must distinguish between each different audit record type in translating an audit file into NADF format. Moreover, the physical binary layout of native audit files sometimes does not respect the specification provided in the documentation of the auditing software. As a result, programming a format adaptor may require manual examination of the actual content of audit trails and is an "try and error" work. This imposes a considerable burden on people willing to port ASAX to a wide range of platforms. Indeed, audit trails are often considered as the bastard of computer security [57].

From our experience on developing format adaptors for several platforms such as SunOS. 4.3, BSM Solaris 2.5, SINIX, and BS2000, we retained two requirements for the format adaptor generator. The *power* requirement means that FDL is theoretically able to describe any audit record structure at the price of user-provided decoding routines. *fagen* allows the user to provide his/her own C-routines for decoding the part of the record which is not directly describable using the format description language supported in *fagen*. The *convenience* requirement means the ability of *fagen* to generate format adaptor source code corresponding to the most common audit record structures. The latter almost always (SunOS.4.3, SINIX, and BS2000) consists of a fixed part and a variable part. The fixed part records the audit data that are common to all audit records and includes fields such as the event type, the time stamp, the subject attributes, etc. The variable part comprises the audit data fields that are event type dependent.

The convenience requirement also means the ability to:

- Ignore certain audit record fields because they are either not relevant to audit trail analysis or only serves a technical purpose such as easier parsing of the record structure. Examples of such fields include separator fields, length fields, number of items in a list, etc.

- Store certain fields under a different a representation which is more suitable for audit trail analysis. For instance, it could be desirable to transform user ids to the corresponding user names, the binary representation of the time stamp to a formatted ASCII string, or to transform IP addresses to the corresponding host name, etc.

- Aggregate several fields in the native audit record into a single one. For instance, the user and system cpu times may be mapped to a single field which is the total cpu time.

The above capabilities are especially useful in a heterogeneous environment for reconciling different audit record sources. Ideally, the transformation of various audit record sources should yield a unique audit record structure for the audit trail analysis.

11.3 Design Choices

In order to meet the convenience requirement, *fagen* uses a language à la C for describing the structure of the input audit files. Auditing subsystems generally make available the audit record structure description as well as other utility function definitions through C include files. For instance, the token structure description in BSM 2.5 are contained in the file `/usr/include/bsm/audit.h`, which has a C definition for each token type. Therefore, using a format description language à la C avoids the effort of learning a new description language and in most cases the format description which must be given to *fagen* is a slight variation of the C include file. The basic idea in generating an audit data description file from a native format description is to consider all the elementary field names in the description of the native audit file and to generate an (NADF) audit data name for each of them. However, this approach is not satisfactory because of the audit record representation change requirement. To get closer to our objectives, we introduce an intermediate description of the audit file which is:

- whose syntax is close to C (but is more elaborate than C),

- which allows us to introduce, eliminate, and aggregate data from the native audit file,

- the elementary fields of which are the actual (NADF) data fields.

In other words, the native record description, henceforth referred to as the *source description*, allows us to define the structure layout of the native file, whereas the intermediate description, henceforth referred to as the *target description*, allows us to define the structure layout of the NADF file. Thus a target description specifies the following three items:

1. A variable data structure à la C (note that in C the format is not fully specified and the actual format is known only at run-time). For instance, in C, an array definition only specifies the maximum number of array elements and the actual number of elements is only known at run-time. Similarly, the actual fields that are actually stored in a union is only known at run-time. Consequently, the target description differs from actual C in that it makes the record structure description completely specified; otherwise, the generation of the audit data description file is not possible.

2. A finite set of (NADF) field names. These are the names that will be used in the RUSSEL programs for analyzing the NADF files. Actually, the second point is obtained as a result of the first one (names of the elementary fields, i.e., the leaves in the variable data structure).

3. The rules allowing one to build the value of the fields in the target description from the values of the fields in the source description.

11.4 Informal Description

The format adaptor generator *fagen* is based on a format description language à la C, which is used to define the structure of the native audit file and the structure of the NADF file. There are two steps involved in generating the source code of a format adaptor. In the first step, the user specifies in FDL the format of the native audit records which is a C-like data structure definition. This definition may use other type definitions and for this purpose, exactly as in C, the user has the ability to define other sub-types, macros, and global variables. The latter may be in the same file or may be in include files. To distinguish between the data structure definition of the native audit records and the other C-definitions, the definition of the native audit record is enclosed within the character string "@@". The file containing the format description in FDL is called the *format description file* and has the extension `.fdf`. In the second step, the FDL compiler, *fagen*, is invoked with the format description file as an argument using the following command line:

```
fagen descript.fdf addf fa.c
```

where `descript.fdf` is the format description file, `addf` is the audit data description file name to be generated, and `fa.c` is the name of the C-source file of the format adaptor to be generated. (When the latter two arguments are omitted, *fagen* generates the audit data description file and the format adaptor under the names `addf` and `adaptor.c`, respectively.)

The above described usage of *fagen* is satisfactory provided that the user needs to translate the native records as they are, i.e, all fields appearing in the native records are translated into NADF and their representation is also conserved in the NADF format. However, the user may be interested only in some selected fields in the native records or may want to add new fields, change the representation of existing fields, aggregate several fields into a single one, etc. Thus, for the sake of convenience, a format description file optionally declares a *target description*, which is also delimited by the character string "@@". Similar to the source description, FDL also uses a C-like syntax for the target description. However, the source and target description serve two different goals. Whereas the source description allows the format adaptor to correctly parse the binary layout of the native records, the purpose of the target description is to specify the list of field names that must appear in the converted audit record and how their values are computed from the native record fields. For user convenience, the target description is not a flat list of field names and their values but is also a C-like data structure definition. This has the benefit of allowing the user to implicitly assign the same value for selected fields as in the native audit record.

11.4.1 Initializers and clean-up routine

Usually, before translating an audit record, the user may need to do some housekeeping computations such as updating global data structures or positioning pointers inside the input audit record buffer. So the format description language allows the user to optionally specify routines that must be called prior to the translation of any audit record. Similarly, FDL allows the user to specify which function must be called before the start of the whole conversion process (actually, right after reading the very first audit record). Also, it is possible to specify which function must be called at completion of the whole conversion process in order, for instance, to release auxiliary buffers and close some auxiliary files that are used during conversion.

11.4.2 Value Specifications

Both the source and target descriptions may be viewed as a tree-like structure where each field is represented as a node in the tree and the child nodes of a composite field represent the fields it is composed of. The default value of a field in the target description is the value of the field in the source description which has the same name. Additionally, the user can explicitly specify the value of a field in the target description in three different ways.

- As a C-valid expression containing no function calls. In this case, the value can either be a constant, a field name in the source declaration, a global variable name, or a C-expression involving a combination of these.
- As a value returned by a user-provided function. The latter is passed the address of the input audit record in order to allow the function to use the fields in the input audit record.
- Instead of computing the value of a field in the target description, it is also possible to specify a procedure call which directly stores the value of a field in the NADF record. In this case, the procedure is passed the address of the input audit record as well as the current position in the NADF buffer at which the computed value must be stored.

Consequently, we provide the user with constructs in the FDL language that allow him/her to express how the value of an audit data field in the target record is computed from various other data fields. These constructs use not only the native record, but also global variables, constants, and possibly user-provided routines. As a result, the user is theoretically able to use the full expressiveness of the C-language. However, these expressions cannot reference fields in the target description itself in order to avoid cyclic references.

The syntax of a target description is the same as a data structure definition in C with two differences: array bounds and union definition. An array bound can be specified in two different forms: $[m]$, in this case the array is assumed to always have exactly m elements, where m is a constant. $[l{:}m]$, where l is an integer expression and m is an integer constant. In the latter case, the array is assumed to have at most m elements and the actual number of elements is determined by the integer expression l, the value of which is determined at run-time.

In a target description, a union is declared as

union tag **switch** (*discriminant*) {
  val_1: $declaration_1$; ... val_n: $declaration_n$; [**default**: *declaration*;]
}

where *discriminant* is a C-valid arithmetic expression, val_1, ..., val_n are arithmetic constants, and $declaration_1$, ..., $declaration_n$ are declarations. When the discriminant is evaluated to val_i ($1 \leq i \leq n$), $declaration_i$ is assumed; otherwise, the default declaration *declaration* is assumed. As a result, the generated format adaptor source code has the ability to figure out which declaration is actually used in a native record.

11.5 Syntax of FDL and Examples

In this section, we provide examples illustrating the BNF syntax of the format description language of Section C.1. These examples are illustrative and are not intended for real-file applications of FDL; Section C.2 contains a realistic example of using FDL and *fagen*.

11.5.1 Example 1

In this example, we illustrate how to eliminate selected fields in the source description, how to change the representation of certain other fields, and how to aggregate fields. Of the audit fields of the source description of Figure 11.1, only the fields auid, rgid, pid are retained in the target description. Moreover, we change the representation of the event type from the decimal representation to the ASCII string event_name, which is the corresponding event name (e.g., the event name identified by the number 1 is represented as the ASCII string 'EXIT'). The value of the field event_name is directly stored in the NADF record by the procedure *getEventName*(), which is passed the address of the native record and the address in the NADF record buffer at which the event name must be stored. *getEventName*() maps the event type to the corresponding event name using the file `/etc/security/audit_event`, which stores such a mapping under BSM 2.5. Similarly, the target description stores the formatted string representation of the time stamp field `time_stamp` into the field `fmt_time` using procedure *getTime*(). Additionally, the fields `user_time` and `system_time` are aggregated into a single field `total_time`, which is the sum of the two.

11.5.2 Example 2

In this example, we illustrate the use of unions and also how the value of certain fields can be retrieved from global variables. The source description given in Figure 11.2 defines the native record, where the fixed part is composed of the event type `event_type`, the audit user id `auid`, the group id `gid`, the process id `pid`, and the time stamp `time_stamp`. The variable part, which is a union containing either the attributes of a `CREAT` event or of a `CHMOD` event. The attributes of a `CREAT` event are: the pathname of the created file represented as a character array of a maximum length MAXPATHLEN, while the actual length is represented by `pathnamelen`, and the file system id `filesystemid` of the created file. The `CHMOD` event stores, in addition to the same attributes as the `CREAT` event, both old and new permission modes. In the target description, the actual event

```
@@
struct {
   int ruid;
   int auid;
   int rgid;
   int pid;
   int event_type;
   int user_time;
   int system_time;
   long time_stamp;
} rec;
@@
@@
struct {
   int auid;
   int pid;
   int rgid;
   char event_time[] ::= getEventName;
   char fmt_time [] ::= getTime;
   int total_cpu_time ::= user_time + system_time;
} rec;
@@
```

Figure 11.1: Example 1

represented by the union in the source description is made explicit by the discriminated union. The latter indicates, for each possible value of the union discriminant (`event_type`), the actually stored event attributes. In the target description, the character array `pathname` declares both the actual length `pathnamelen` (which do not appear as a field in the NADF record) and the maximum length `MAXPATHNAMELEN`. For the `CREAT` event, the representation of the field `perm_mode` is changed to a formatted string `fmt_perm_mode`. The latter is a character array computed, element by element, from the global array `fmt_mode[]`. The content and actual length of `fmt_mode[]` are initialized by call to the function *putFmtPermMode*(), which is invoked each time a new native record is input using the construct **to step**. For example, to the decimal representation of the permission mode 755 corresponds the formatted string OWNRWX-GRPRX-OTHRX. For the `CHMOD` event, the same formatted string representation of the permission mode `old_mode` and `new_mode` are directly stored in the NADF record by the procedure *getFmtMode*().

11.5.3 Example 3

In this example, we illustrate the use of the constructs **at start**, **to step**, and **at end** for performing a simple statistics on the total number of records per event type. To the previous example, depicted in Figure 11.2, we add the following three lines

```
at start initEventStatTable();
to step updateEventStatTable();
at end printEventStatTable();
```

The functions *initEventStatTable*(), *updateEventStatTable*(), and *printEventStatTable*() manipulate the global statistic table `event_state_table[]` defined in Figure 11.3.

When the native audit file conversion starts, and before the very first native record is input, *initEventStatTable*() is called to initialize the table `event_stat_table[]`; the definition of the three above functions as well as the definition of the table `event_stat_table[]` are depicted in figure 11.3. Every time a new native record is input, and before it is converted, *updateEventStatTable*() increments in the corresponding entry of `event_stat_table[]`, the accumulated total number of events for the event type represented by the audit record. Upon termination of the conversion, i.e, when the end of the native audit file is reached, *printEventStatTable*() reports the content of the statistic table `event_stat_table[]`.

11.6 Translation Rules

The semantics of the format description language is defined by translation rules, which allow us to mechanically generate the audit data description file as well as the source code of the format adaptor. Starting from the source and target descriptions contained in the format description file, we first use the translation rules in order to derive an internal representation of the described format in the form of a declaration tree. Such a declaration tree is a reformulation of the target description, where each node corresponds to a sub-declaration and where the sub-declarations of elementary fields are represented by the leaves of the tree. Based on the declaration tree, we generate the audit data description file simply by enumerating all the possible paths from the root node to the leaves of the tree. The declaration tree is subsequently added other information items necessary for generating the source code of the format adaptor. Section 11.6.1 presents how a declaration tree is produced from the target description. Section 11.7 describes how an audit data description file is

```
#define MAXPATHNAMELEN 512
@@
struct {
   int event_type;
   int auid;
   int pid;
   int time_stamp;
   union {
      struct creat {
         char pathname[MAXPATHNAMELEN];
         int pathnamelen;
         int filesystemid;
         int perm_mode;
      } creat_file;
      struct chmod {
         char pathname[MAXPATHNAMELEN];
         int pathnamelen;
         int filesystemid;
         int oldmode;
         int newmode;
      } chmod;
   } un_rec;
} rec;
@@
@@
struct {
   int event_type;
   int auid;
   int gid;
   int pid;
   long time_stamp;
   union switch (event_type) {
      case 1:  /* creat */
         struct {
            char pathname[pathnamelen:MAXPATHNAMELEN];
            char fmt_perm_mode ( = fmt_mode[i1])[fmtlen];
         } creat_file;
      case 2 :  /* chmod */
         struct {
            char pathname[pathnamelen:MAXPATHNAMELEN];
            char fmt_old_mode [] ::= getFmtMode;
            char fmt_new_mode [] ::= getFmtMode;
         } chmod;
   } un_rec;
} rec;
@@
to step putFmtPermMode();
```

Figure 11.2: Example 2

```
#define MAX_EVENT_NBRE 280

struct event_stat {
   int event_type;
   int total_nbre;
};
typedef struct event_stat event_stat_t;

event_stat_t event_stat_table[MAX_EVENT_EVENT_NBRE];

at start initEventStatTable();
to step updateEventStatTable();
at end printEventStatTable();

initEventStatTable()
{
   int i;
   for (i = 0; i < MAX_EVENT_NBRE; i++) {
      event_stat_table[i].event_type = i;
      event_stat_table[i].total_nbre = 0;
   }
}

updateEventStatTable()
{
   event_stat_table[rec.event_type]++;
}
printEventStatTable()
{
   int i;

   printf("EVENT STATISTICS \n");
   printf("Event type Total number\n");
   for (i = 0; i < MAX_EVENT_NBRE; i++) {
      printf("%d|%d\n", event_stat_table[i].event_type, total_nbre);
   }
}
```

Figure 11.3: Example 3

generated based on the produced declaration tree. In Sections 11.8.1 and 11.8.2, we determine how the fields in the target description are expressed in terms of the fields in the source declaration as well as global variables.

11.6.1 Declaration Tree

In this section, we map the syntactical structure of the target description onto a tree structure, called the *declaration tree*, which is better suited for our discussion. The declaration tree is constructed recursively on the components of the target description.

Let us first introduce some definitions. The *identifier* of a declaration is the name being declared. For instance, the identifier of `int i;` is i while the identifier of `struct {int i; char name[4];} data[5];` is *data*. The declaration of the native record and the target audit record declaration are referred to as the *root source declaration* and the *root target declaration*, respectively. Any sub-declaration of the root source (resp. target) declaration is also referred to as a source (resp. target) declaration. Note that a target declaration can always be written as:

type identifier $b_1 \ldots b_k$ <value_spec> $b_{k+1} \ldots b_n$;

where

- $0 \leq k \leq n$
- $b_i = [l_i : m_i]$ or $[m_i]$, where l_i is an arithmetic expression determining the actual length of the array and m_i is an integer constant determining the maximum length of the array.

If $k = 0$ the sequence of array bounds $b_1, \ldots, b_k$ is empty and similarly, if $k = n$, the sequence of bounds $b_{k+1}, \ldots, b_n$ is empty. A target declaration is said to be *annotated* if its value specification <value_spec> is not empty; it is said to be *unannotated* otherwise. For instance, the declarations `int i;` and `struct {int i = 2; short int k = 3;} s[2];` are unannotated whereas `int i = 2;` and `short int k = 3;` are annotated. A declaration is said to be *compound* if its type is a union or a struct and if it is unannotated; it is said *simple* otherwise.

Under the above definitions, we associate a declaration tree with the root target declaration recursively as follows:

1. with each sub-declaration *type name* $b_1 \ldots b_k$ <value_spec> $b_{k+1} \ldots b_n$; $(0 \leq k \leq n)$ is associated a node in the declaration tree which is tagged with the indexed variable name $name[i_1][i_2] \ldots [i_{n-k}]$, where $0 \leq i_j \leq m_j$ and $b_j = [l_j{:}m_j]$ $(k+1 \leq j \leq n)$.

   In case the type is `char`, the tag is $name[i_1][i_2] \ldots [i_{n-k-1}]$. $m_{k+1} * \ldots * m_k$ is called the *multiplicity* of the node.

2. the root target declaration is associated with the root node of the declaration tree.

3. if a target declaration is a simple declaration, its associated node has no descendents.

4. otherwise, if the type is a compound type a target declaration is:

   (a) `struct` $\{d_1 \ldots d_n\}$ where $d_1, \ldots, d_n$ denote declarations, then its associated node has n descendent nodes corresponding to $d_1, \ldots, d_n$.

   (b) `union switch (discriminant)` $\{$ `case` c_1 `:` d_1`; ...; case` c_n `:` d_n`; [default :` d_{n+1}`;]`$\}$, then its associated node has n ($n+1$ if the default case is present) descendent nodes corresponding to $d_1, \ldots, d_n$ ($d_1, \ldots, d_{n+1}$ if the default case is present).

Note The exception rules for the type `char` are meaningful because when a user declares an array of type `char`, s/he almost always wants to consider the field as a unit i.e., the array must appear as a single field in an audit record; s/he seldom wants each character to appear as a separate field of its own. So if a node is of the form `char name[s:m]` it is a leaf in the declaration tree and in addition, its tag is *name* and not *name*[i$_i$].

11.7 Generation of the Audit Data Description File

Based on the declaration tree constructed in the previous section, we show how one can systematically generate all the possible field names as they appear in the audit data description file. The basic idea in constructing these field names is to perform an enumeration of all the path leading from the root node to a leaf following a depth first traversal of the declaration tree. Every time we reach a leaf, we concatenate all the tags found along the traversed path. Its is important to note that the so-generated field names are the ones which will be used to reference the fields of the audit records in a RUSSEL module. Moreover, we also generate the source type and the target type of each field. The latter is the type of the target declaration associated with the leaf node in question. By contrast, the source type is only generated when possible i.e., when the field name involved can be mapped back to a field in the source declaration which has the same name (see Section 11.8.1). For instance, when a field is added to the target declaration, it cannot be mapped back to a field in the source declaration. The source and target types are useful for correctly decoding the value of a field during audit trail analysis.

In the following, we present the rules used to generate the field names, their source and target types, and the audit data identifiers.

1. Audit data names are constructed by concatenating the sequence of tags found in the declaration tree along the path from the root node to a leaf. Since all obtained names begin with the prefix 'rec.', this prefix is redundant and is consequently removed, except in the extreme case where the declaration tree is restricted to its root node. In this case, there is a single audit data which is the whole audit record. Moreover, nodes with tags of the form *name*$[i_1] \ldots [i_n]$ where $0 \leq i_k \leq m_k - 1$ $(k = 1, \ldots, n)$ are considered as being $m_{k+1} * \ldots * m_k$ nodes whose tags are *name*$[0] \ldots [0]$, ..., *name*$[0] \ldots [m_n - 1]$, ..., *name*$[m_1 - 1] \ldots [m_n - 1]$. For instance, suppose that the sequence of the tags found are: `rec`, `arg`, and `openFiles`$[i_1]$ $(0 \leq i_1 < m_1 = 3)$, then we obtain 3 field names which are: `arg.openFiles[0]`, `arg.openFiles[1]`, and `arg.openFiles[2]`.

2. The audit data ids are generated simply by enumerating the field names obtained in (1) from 1 to N, following a depth first traversal of the declaration tree.

3. The audit data types are generated as follows. Recall that it is not always possible to relate a field name in the source declaration. This is simply because new fields may be defined in the target declaration and hence does not have a relative in the source declaration.

Example The audit data description file corresponding to example 2 after generating the declaration tree is as follows (written in two columns):

```
1 1                          1 5
2 int                        2 undef
```

```
3 char[]
4 usrname
5
1 2
2 int
3 char[]
4 grpname
5
1 3
2 int
3 int
4 pid
5
1 4
2 undef
3 char[]
4 arg.arg_sub.filename
5
```

```
3 int
4 arg.arg_sub.mod
5
1 6
2 undef
3 char[]
4 arg.filename
5
1 7
2 int
3 int
4 even
5
1 8
2 long
3 long
4 time
5
```

11.8 Determination of default and fully expressed value specifications

The purpose of this section is to explain how the value of the audit data (of the NADF file) are computed from the native audit record. Let us first introduce the problem through an example.

Example Consider the following format description file:

```
int  y = 123;
@@
struct {
     int x;
     struct {
           int x;
           char c;
           long l;
     } s;
} rec;
@@
@@
struct {
     int x = y;    /* 1 */
     struct  {
           int x = x + y;     /* 2 */
           int y = x;         /* 3 */
    } s;
```

```
} rec;
@@
```

It is not clear how one can relate the variable names `x` and `y` in the value specifications 1, 2, and 3 to a source declaration. Thus, we have to fix a systematic way to derive the default value specification from the structure of the target record declaration. In fact, the fully expressed value specification in this example are as given below.

```
@@
struct {
     int x = y;    /* the global variable y */
     struct  {
           int x = rec.s.x + y;
           int y = rec.s.x;
    } s;
} rec;
@@
```

The expression `rec.s.x` is called the *fully expressed value specification* corresponding to the value specification ' = x '.

From the previous example we see that the value of a field in the target description can be expressed using relative value specifications instead of fully expressed value specifications. For instance, the relative value specification of the field (3) in the previous example is `x`, whereas its fully expressed value specification (or absolute value specification) is `rec.s.x`. Using only relative value specifications is more convenient for the user because this yields a more concise code. The fully expressed value specification is implied by the "context" of the declaration of the field involved. Intuitively, the target declaration and the source declaration often have similar syntactic structures. The outer most block structure is composed of both source and target declarations as well as all declared global variables, function prototypes, and constants. Assuming that the source and target declarations are both of struct type, we define a new block structure which is composed of the struct elements. By walking the source and target descriptions top-down, a new (inner) block structure is defined every time two elements of the current block are found to have the same declared name and are of the type struct or union. The inner most block is one with all fields of basic types.

The above scheme suggests a hierarchical context composed of the sequence of local contexts where each local context contains the set of declarations in a block structure. To relate a given field name in the target declaration, we first look up this name in the current local context. If one is found, it is related to it; otherwise, we look up the name in the outer context, and so forth. For instance, considering the previous example, the fields of the inner most block structure declaration in the target declaration are associated with the hierarchical context $\mathcal{C} = \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$, where

$\mathcal{P}_1$ = {`int y = 123;`, `struct {int x; struct {int x; char c; long l;} s; } rec;` },
$\mathcal{P}_2$ = {`int x;`, `struct {int x; char c; long l;} s;`},
$\mathcal{P}_3$ = {`int x;`, `char c;`, `long l;`}.

Formally, a hierarchical context is defined as follows.

Definition 9 A context $\mathcal{C}$ is a possibly empty sequence of sets $\mathcal{P}_1, \ldots, \mathcal{P}_n$, called *local contexts*. A local context is a set of variable declarations. A variable v is said to be *active* with respect to the context $\mathcal{C}$ if and only if there exists some i $(0 \leq n)$ such that $v \in \mathcal{P}_i$ and $v \notin \mathcal{P}_j$ for $j = i+1, \ldots, n$. Otherwise, the variable v is said to be *inactive* with respect to $\mathcal{C}$.

It follows from the above definition that every context $\mathcal{C}$ has at most one active variable of a given name. The expression "*the variable x in $\mathcal{C}$*" stands for the unique variable name x which is active with respect to $\mathcal{C}$.

Let G be the set of global variable declarations (including the root source record and excluding the root target declaration), C the set of defined constants, and F the set of function declarations in the format description file. Each node d in the declaration tree associated with the target declaration is uniquely associated with a context, denoted by $\mathcal{C}(d)$ as follows:

1. the root node in the declaration tree is associated with the context $\mathcal{C}_0$ consisting of the single local context $\mathcal{P} = G \cup C \cup F$.

2. Let d be a compound node in the declaration tree and *name* its associated tag. If *name* is inactive with respect to the context $\mathcal{C}(d)$, then the context associated with every descendent node of d is also $\mathcal{C}(d)$. Otherwise, let SD(d) be the active declaration in $\mathcal{C}(d) = \mathcal{P}_1, \ldots, \mathcal{P}_n$ and j such that $1 \leq j \leq n$, SD$(d) \in \mathcal{P}$, and SD$(d) \notin \mathcal{P}_i$ for $i = j+1, \ldots, n$. We determine the context associated with every descendent node of d as follows:

   (a) if SD(d) is of the form:

   $$\texttt{struct}\ \{d_1 \ldots d_m\}\ \mathit{name}\ b_1 \ldots b_r; \qquad m \geq 1, r \geq 0$$

   then the context of every descendent node of d is:

   $$\mathcal{C}' = \mathcal{P}_1, \ldots, \mathcal{P}_j, \mathcal{P} = \{d_1, \ldots, d_n\}.$$

   (b) Similarly, if SD(d) is of the form:

   $$\texttt{union}\ \mathit{tag}\ \texttt{switch}\ (\text{intExpr})\ \{\ \texttt{case}\ c_1 : d_1 \ldots \texttt{case}\ c_m : d_m\ [\texttt{default} : d_{m+1}]\ \}\ \mathit{name}\ b_1 \ldots b_r; \qquad m \geq 1, r \geq 0$$

   then the context of every descendent node of d is:

   $$\mathcal{C}' = \mathcal{P}_1, \ldots, \mathcal{P}_j, \mathcal{P} = \{d_1, \ldots, d_n\}.$$

   ($\mathcal{P} = \{d_1, \ldots, d_{m+1}\}$ if the default case is specified.).

   (c) otherwise, if SD(d) uses a basic type or an array of basic type, then the context of every descendent node of d is also $\mathcal{C}(d)$.

11.8.1 Determination of default Values

This section presents an algorithm that calculates the default expression of a field assuming that its declaration does not specify a value specification. In other words, given a declaration tree augmented with contexts and a leaf node with an empty value specification, the purpose of the algorithm is to derive the default expression of this field. This pseudo-algorithm is depicted in Figure 11.8.1.

The procedure *identifier*(n) returns the identifier of the node n, *context*(n) returns the context attached to node n, *level*(C) returns the level of the context C, i.e., the length of the sequence of local contexts comprising the context. The operator `<>` yields the concatenation of its operands, and Pn(C) returns the n-th local context of C.

```
default(node)
begin
   n := leaf;
   name := identifier(n);
   C = context(n);
   suffix := '';
   while (level(C) > 1 and n != root) do
   begin
      if (name in Pn(C))
         begin
            if (suffix = '') suffix := name;
            else suffix := name <> '.' suffix;
            n := parent(n);
            C := context(n);
            name := identifier(n)
         end
      else
         begin
            n := parent(n);
            C := context(n);
         end;
   end;
   if (name in Pn(C))
      if (suffix = '') suffix := name
      else suffix := name <> '.' <> suffix
   else
      Error;
end.
```

Figure 11.4: Algorithm for calculating the default expressions.

Example Consider the following format description file given below.

```
int y = 123;
@@
struct {
    int x;
    struct {
        int x;
        char c;
        long l;
    } s;
} rec;
@@
@@
struct {
    int x; /* 1 */
    struct {
        int x; /* 2 */
        int y; /* 3 */
        int z; /* 4 */
    } s;
} rec;
@@
```

In this case, the execution of the algorithm of Figure 11.8.1 produces the following results: `rec.x, rec.s.x, y` for nodes 1, 2, and 3, respectively. The algorithm reports an error on node 4.

11.8.2 Fully Expressed Value Specifications

The purpose of the present section is to derive the fully expressed value specification from a given value specification. Basically, a value expression of a given node is expanded using the context attached to the given node. There are three cases to consider according to the three possible forms of value expressions: = *expression, := function, and ::= procedure.* If *expression* is a constant, its fully expressed value specification is itself; if *expression* is an identifier, its fully expressed value specification is obtained by applying the algorithm of Figure 11.8.1 on the identifier. If *expression* is an indexed name of the form $name[i_1]\ldots[i_n]$, we derive the fully expressed value specification of *name* considered as a simple variable name, let *full_path_name* be the fully expressed value specification of *name*, the the fully expressed value specification of $name[i_1]\ldots[i_n]$ is $full\_path\_name[i_1]\ldots[i_n]$. If *expression* is an arithmetic expression, we first determine the fully expressed value specification of each operand and we apply the operators on the fully expressed value specifications obtained. If the value specification is of the second form, i.e., `:= function`, we evaluate *function* with respect to the context $\mathcal{C} = \mathcal{P} = \mathrm{F}$, where F denotes the set of function declarations in the format description file. If *function* is active in $\mathcal{C}$, the fully expressed value specification is

$$function(\&\mathrm{rec}, i_1, \ldots, i_n)$$

where `&rec` is the address of the native record and k is the dimension of the declared array or is 0 if the node declares a simple variable. This means that the value of the field in the NADF record is returned by the function call above. If the value specification is of the third form, i.e.,

`::= procedure`, we look up the procedure name in the set of declared functions as in the second form and the fully expressed value specification is

$$procedure(\&\text{rec}, \&\text{crt\_pos}, i_1, \ldots, i_n)$$

where &*crt_pos* is the current position in the NADF buffer at the time the involved field is being converted to NADF format; (&rec, $i_1, \ldots, i_n$, have the same meaning as in the second form. Finally, the fully expressed value specification of a union discriminant is obtained exactly as in the case of a value specification of the first form.

11.9 Source code generation

For each basic type (i.e., char, int, long, etc.) we provide a conversion routine which stores a value of this type into NADF format. Such a conversion routine receives the audit data id, the audit value, and the current position in the NADF record buffer. The audit data id is as determined in Section 11.7 and the audit data value was determined in Sections 11.8.1 and 11.8.2. To convert a struct declaration into NADF format, we sequentially convert each component in the struct into NADF format. To convert a union, we use a switch statement on the union discriminant to convert each possible component. Finally, to convert an n-dimensional array of fields of the same type, we use n embedded for loops, where each loop ranges from zero to the size of the corresponding dimension.

A full example illustrating the use of *fagen* is given in Section C.2, which is concerned with generating the source code of a format adaptor for converting *utmpx* accounting files. The example depicts the format description file, the source code of the format adaptor and the audit data description file.

11.10 Conclusions and Limitations

This chapter presented a format adaptor generator tool which reduces the cost of implementing format adaptors for the NADF format. The format adaptor generator uses a format description language à la C which is convenient for describing common audit record layouts. It also provides the user with the ability to eliminate, change the representation, and aggregate fields in the native format, thereby allowing the user to retain only the fields of interest and change their representations if needed. The language is powerful enough to cover other layouts which are not directly expressible in the language using user-provided routines. We presented a full example showing the usefulness and effectiveness of the format adaptor generator in a real-life application.

A shortcoming of the language is that it cannot directly describe sequences of data items, i.e., a contiguous sequence of data elements of different types (e.g., a `char` immediately followed by an `int`). Note that the latter data type is not available in C either. In particular, the struct data type is not adequate for representing sequences since it introduces padding bytes between elements of the sequence in order to ensure proper alignment of the struct components. For instance, in C, the binary layout of the declaration `struct { char c; int i;}` consists of a character followed by three padding bytes followed by four bytes representing the `int`. The introduced padding bytes ensure the proper alignment of the integer and are machine and compiler dependent.

Another shortcoming of the generator is its inability to handle the case where the format adaptation is performed on a machine architecture with different byte orders and alignments restrictions than the machine on which the audit trail was generated. Handling this problem requires a prior conversion of the binary representation of the audit trail from one format to another according to the byte orders and alignment restrictions of the involved machines. This conversion process is commonly referred to as "marshaling" and is involved, for instance, in converting IPC arguments from the sender's local format to the receiver's local format. This may require swapping bytes, adjusting alignments, changing the size of certain values, etc. This marshaling process must be done, for instance, in a distributed audit trail analysis, where the audit records transmitted from a slave node must be converted to the central evaluator binary format. We consider the audit record marshaling problem as a particular case of the general marshaling problem of communication protocols. Such a problem is addressed in the literature and one example tool performing this task is USC (Universal Stub Compiler) [58]. USC presents a general solution to this problem that automatically generates stubs to convert a C data structure with one user-defined format to a C data structure with another user-defined format. A future work is to couple USC with our *fagen* such that the source code generated by *fagen* uses the stubs generated by USC for the source description.

Chapter 12

Conclusions and Future Works

12.1 Conclusions

The work presented in this thesis constitutes an original contribution to the intrusion detection research effort. The major contribution of the thesis is the design and the implementation of the rule-based language RUSSEL, which embodies control structures that are inherently targeted towards powerful and convenient analysis of event streams in general and security audit trails in particular. In designing the language as well as the architecture of the system, we were committed to meeting fundamental requirements, i.e., power, efficiency, and universality. These requirements are fully met as demonstrated by the real-life example detection modules for various platforms (UNIX, SINIX, and BS2000) and by performance evaluations. Other goals such as graphical user interface and enhanced reporting capabilities are considered not as critical as the goals pursued in this thesis.

Integrating intrusion detection with configuration analysis allowed us to enhance and tune our detection tool and enabled us to report misconfiguration-related vulnerabilities on-line. As far as we know, there is no system in the intrusion detection literature that integrates both intrusion detection and configuration analysis in a single unified architecture.

The thesis also presented a distributed architecture and a prototype system for intrusion detection in a distributed heterogeneous environment. The implemented prototype is a proof of concept which demonstrates the feasibility and practicality of the presented requirements. The distributed system uses a hierarchical master-slave architecture, which is suitable for achieving a scalable system, thereby enabling increasing levels of event abstractions. Furthermore, we demonstrated the efficiency of the system by performance measurements.

Additionally, we tackled the task of developing a format adaptor generator which allows us a rapid development of format adaptors; in the most common cases, this development cost is virtually eliminated.

12.2 Future Works

In this section, we outline some research directions towards further enhancements of the work presented in this thesis.

12.2.1 The Language RUSSEL

We see three main improvements to RUSSEL.

Higher-Level Language à la SQL

It could be argued that the effective use of ASAX requires learning the language RUSSEL; because RUSSEL uses a mixture of imperative and rule-based paradigms, its programming style could be difficult to grasp for non-expert users. Based on this observation, it would be useful to design a higher-level language à la SQL, which allows the user to express more declarative queries on the audit trail. These queries could have the form of regular expressions defining event patterns. The basic methodology of programming the detection of regular event patterns in RUSSEL is an important step towards the goal of designing the intermediate language. Notice that, the higher-level language is likely to be less expressive than plain RUSSEL, but it has the advantage of being accessible to less skilled security administrators.

Rule Triggering Optimization

The rule triggering mechanism induces a non-negligible performance cost, which is mainly related to allocating a dynamic memory area for storing the representation of the rule instance. Also, the cost of copying actual parameter values of the triggered rule instance can be performance consuming especially if the parameters are long strings. Additionally, after a rule instance has been executed, the memory area allocated for it must be released. A large class of RUSSEL rules are designed in such a way that certain rules are triggered with the same parameter values. This is due to the fact that event-patterns almost always specify that the sequence of events comprising the pattern can be separated by irrelevant events. So typically, a rule ignores these intervening events and simply retriggers itself with the same parameters. Thus, a useful optimization in this specific case is to reuse the same dynamic area, thereby eliminating the overhead of allocating, initializing, and releasing this dynamic area. This optimization can be very rewarding considering the fact that in most cases, a large number of rule instances of the same rule may be retriggered over and over again. This optimization requires a reliable means of verifying whether the parameters of the triggered rule have been altered or not. Such a problem can be addressed by static analysis [59], which is the basis of modern compiler optimization techniques. These techniques must however be adapted to the specificity of RUSSEL.

Layering Rule Instances: Co-routining

A further important optimization of RUSSEL can be achieved by considering several layers of event streams instead of a single layer which is the audit trail being analyzed. The basic idea in this optimization is to have n layers of active rule instances; each layer being executed against a stream of records of the same layer. Basically, when a record is filtered by a rule instance belonging to a given layer, it is copied or passed upward, to the second higher level layer and so on. Rule instances in a given layer analyze only the audit record stream belonging to the layer of the same level. The advantage of this optimization is that, in the average, only a small subset of active rules are executed against the current record. Relevant records are successively pushed up the layered stream and undergo an increasingly detailed analysis. By contrast, irrelevant records are directly discarded by lower layer rule instances, thereby sparing unnecessary execution of rules instances belonging to higher-level layers. Considering the example depicted in Figure 4.1, the rule *count_rule* is executed against every record in the audit trail. Using the optimization scheme outlined above, we would have two layers of rule instances and correspondingly two layers of event streams. The rule *badlogin* and *count_rule* belong to the layers 1 and 2, respectively. Moreover, the layer 1 of audit records is simply the audit file whereas the layer 2 is exclusively composed of audit records representing login failures and are pushed up the stream using the rule *badlogin*. In this case, the rule *count_rule* is executed only on failed login events. Assuming that only 5% of the total number of audit records represent a login failure, the optimized version executes *count_rule* 95% less often than in the non-optimized version.

As the reader may have noticed, there is a close relationship between the above layering scheme and the master-slave architecture used in the distributed audit trail analysis system presented in Chapter 10. Basically, the hierarchical architecture of the distributed system can be seen as a special form of the above layering, where the rule instances executed by a slave evaluator belonging to level k in the hierarchy allows the system to filter events and transmit them to the master evaluator. The rule instances executed by the master evaluator then belong to level $k+1$. The difference between the optimized version and the distributed audit trail analysis is that the latter is composed of several communicating processes (the master and slave evaluators) whereas the former uses a single evaluator with layered rule instances.

12.2.2 The Integrated System

We see two useful improvements to the integrated system presented in Chapter 9. The first improvement is to extend the datalog program in order to address other potential sources of misconfiguring the protection system. Such potential sources include filesystems exported to unauthorized hosts, filesystems exported with write permissions, etc. The second improvement, alluded to in Section 9.6.3, consists of using the distributed audit trail analysis system to enhance the accuracy of the fact base. To ensure the accuracy of the fact base, we have to take into account all relevant modifications of the fact base which take place in any host that NFS mounts the local filesystem. Relevant audit records reporting fact base modification and occurring in remote hosts are filtered by the local slave evaluators and communicated to the master evaluator which updates the global fact base accordingly.

12.2.3 The Distributed System

There are four useful improvements related to the distributed audit trail analysis system presented in Chapter 10, which we outline in the following sections.

Enhanced Reporting Capabilities

We did not explicitly consider intrusion reporting capabilities as a distinguished requirement during the design of the distributed system architecture. Indeed, a carefully designed activity reporting module must be an integral part of the system because in practice, and especially in case there are dozens of monitored hosts, an overwhelming number of report messages can be very annoying for the security administrator. The latter could not cope with all the messages originating from the monitored hosts and this increases the chances of important alarm messages being overlooked. A straightforward approach is to identify classes among all possible report messages using a severity, or priority, criteria. The security administrator has then the ability to focus only on the most important messages based on their severity. Less important messages are considered only in order to provide added insight for further investigations on reported security violations. Such report messages can be represented as a special form of audit records, which are tagged accordingly to be distinguishable from normal audit records. When an activity report record is received by the master evaluator, its content is formatted appropriately and printed on the central console.

Enhanced Availability of the Master Evaluator

Whereas the failure of a slave evaluator does not affect the operation of the rest of the monitored hosts, the failure of the master evaluator requires a manual intervention of the security administrator in order to restart the distributed analysis from another available host. Although the availability requirement ensures that no audit records are lost or missed by the analysis, the manual intervention could be avoided by having the slave evaluators perform a voting procedure to elect a master evaluator among a specified list of slave evaluators. This automatic failure recovery scheme is used for instance in [56] (which is also based on a similar master-slave architecture). During the voting procedure, the process of filtering records at slave hosts is suspended until a new master is elected and then the filtering is resumed by redirecting the filtered event streams to the newly elected evaluator. The latter initiates a console process and at that time, the distributed audit trail analysis is completely recovered.

Network User Identity Tracking

In monitoring user activity across multiple hosts, it is useful to be able to attribute multiple user session activities to the same user even though that user initiated these sessions under different identities. As the user opens new sessions from one host to another, his/her activity should be tracked and attributed to a unique identity under which the user opens the first session in the network environment. Such network user identity (NID) is presented in [60], which describes an algorithm for user identity tracking over a local area network. Our distributed system is able to relate multiple user sessions to a unique NID thanks to the event correlation capability of the system. The basic idea is to correlate two events taking place in two hosts: the host initiating the new telnet or rlogin session and the remote host which is the destination of the telnet or rlogin connection. At the former host, the local evaluator observes a telnet event with the local host name, local user id, the remote host name, and the remote user id. At the latter host, the slave evaluator attached to it also observes a login event with the source host name and the user id of the initiated login session. Both of these two events occurring on both sides of the telnet connection can be easily correlated at the master evaluator, which is then able to figure out that all user activity on the new session must be attributed to the same NID. However, in the above described scheme, we assume that all hosts attached to the local area network are monitored by the distributed system. This assumption is not realistic because it may not be possible to monitor the whole distributed environment since some hosts may not support an auditing mechanism. A more reliable approach which is described in [60] is to analyze the IP packet traffic and monitor telnet connection establishment and reset. An ongoing research related to IP traffic analysis using ASAX is currently pursued which would allow us to implement a similar network user identity tracking algorithm. Due to the broadcasting property of Ethernet local area networks, a single packet "sniffer" is sufficient for monitoring all telnet connections involving any host in the local area network.

Appendix A

BNF Syntax

A.1 BNF Syntax of Tokens

⟨token ⟩ ::= ⟨identifier ⟩
| ⟨constant ⟩
| ⟨special symbol ⟩

⟨identifier ⟩ ::= ⟨letter ⟩
| ⟨identifier ⟩ ⟨digit ⟩
| ⟨identifier ⟩ ⟨letter ⟩
| ⟨identifier ⟩ ⟨underscore ⟩ ⟨letter ⟩
| ⟨identifier ⟩ ⟨underscore ⟩ ⟨digit ⟩

⟨constant ⟩ ::= ⟨integer constant ⟩
| ⟨C_literal ⟩
| ⟨X_literal ⟩

⟨integer constant ⟩ ::= ⟨digit ⟩
| ⟨integer constant ⟩ ⟨digit ⟩

⟨C_literal ⟩ ::=' ⟨C_sequence ⟩'

⟨C_sequence ⟩ ::= ⟨empty ⟩
| ⟨C_sequence ⟩ ⟨C_character ⟩

⟨C_character ⟩ ::= any printable ASCII character except simple
quote (') | ' '

⟨X_literal ⟩ ::= X ' ⟨X_sequence ⟩'

⟨X_sequence ⟩ ::= ⟨empty ⟩
| ⟨X_sequence ⟩ ⟨X_symbol ⟩

⟨X_symbol ⟩ ::= ⟨X_character ⟩ ⟨X_character ⟩

$\langle$X_character$\rangle$ ::= 0 | ... | 9 |A |B |C |D |E |F

$\langle$digit$\rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\langle$letter$\rangle$ ::= a | ... | z | A | ... | Z

$\langle$special symbol$\rangle$ ::= **+** | − | ***** | **(** | **)** | > | < | **!=** | ≤ | | ≥ | **:**
| **%=** | **;** | **=** | −> | **:=** | **,** | **and** | **at_completion**
| **begin** | **div** | **do** | **end** | **false** | **fi** | **for_current**
| **for_next** | **not** | **od** | **off** | **or** | **present** | **rule** | **skip**
| **string** | **if** | **integer** | **mod** | **trigger** | **true** | **var**

A.2 BNF Syntax of RUSSEL

⟨module ⟩ ::= ⟨usage list ⟩
⟨global var decl ⟩
⟨rule decl list ⟩
⟨init action ⟩ **.**

⟨usage list ⟩ ::= ⟨empty ⟩ |
⟨module usage ⟩ **; ...** **;** ⟨module usage ⟩ **;**

⟨module usage ⟩ ::= **uses** ⟨list of modules ⟩

⟨list of modules ⟩ ::= ⟨module name ⟩ **, ...** **,** ⟨module name ⟩

⟨module name ⟩ ::= identifier

⟨global var decl ⟩ ::= ⟨empty ⟩ |
⟨global var group ⟩ **;** **...** **;** ⟨global var group ⟩ **;**

⟨global var group ⟩ ::= ⟨variable class ⟩
⟨variable name ⟩ **, ...** **,** ⟨variable name ⟩ **:** ⟨type ⟩

⟨variable class ⟩ ::= ⟨empty ⟩
| **internal**
| **external**

⟨rule decl list ⟩ ::= ⟨empty ⟩ |
⟨rule declaration ⟩ **;** **...** **;** ⟨rule declaration ⟩

⟨rule declaration ⟩ ::= ⟨rule heading ⟩
⟨variable declaration part ⟩
⟨action part ⟩

⟨rule heading ⟩ ::= ⟨rule class ⟩
rule ⟨rule name ⟩ **(** ⟨parameter list ⟩**)**

⟨rule class ⟩ ::= ⟨empty ⟩
| **internal**
| **external**

⟨rule name ⟩ ::= ⟨identifier ⟩

⟨parameter list ⟩ ::= ⟨empty ⟩
| ⟨parameter group ⟩ **;** **...** **;** ⟨parameter group ⟩

⟨parameter group ⟩ ::=
⟨parameter name ⟩ **,** **...** **,** ⟨parameter name ⟩ **:** ⟨type ⟩

⟨parameter name⟩ ::= ⟨identifier⟩

⟨type⟩ ::= **string**
| **integer**

⟨variable declaration part⟩ ::= ⟨empty⟩
| **var** ⟨variable declaration⟩ **; ... ;** ⟨variable declaration⟩

⟨variable declaration⟩ ::=
⟨variable name⟩ **, ... ,** ⟨variable name⟩ **:** ⟨type⟩

⟨variable name⟩ ::= ⟨identifier⟩

⟨action part⟩ ::= ⟨action⟩

⟨action⟩ ::= **skip**
| ⟨assignment⟩
| ⟨conditional action⟩
| ⟨repetitive action⟩
| ⟨compound action⟩
| ⟨rule triggering⟩
| ⟨predefined procedure call⟩

⟨assignment⟩ ::= ⟨left expression⟩ **:=** ⟨right expression⟩

⟨left expression⟩ ::= ⟨parameter name⟩
| ⟨variable name⟩

⟨right expression⟩ ::= ⟨expression⟩

⟨expression⟩ ::= ⟨constant⟩
| ⟨field name⟩
| ⟨left expression⟩
| ⟨expression⟩ ⟨binary arithmetic operator⟩ ⟨expression⟩
| ⟨unary arithmetic operator⟩ ⟨expression⟩

⟨field name⟩ ::= ⟨identifier⟩

⟨conditional action⟩ ::=
if ⟨guarded action⟩ **; ... ;** ⟨guarded action⟩ **fi**

⟨guarded action⟩ ::= ⟨condition⟩ $\rightarrow$ ⟨action⟩

⟨condition⟩ ::= **true**
| **false**
| **present** ⟨field name⟩
| ⟨expression⟩ ⟨relational operator⟩ ⟨expression⟩

| ⟨condition ⟩ ⟨logical operator ⟩ ⟨condition ⟩
| **not** ⟨condition ⟩

⟨relational operator ⟩ ::= $<$ | $>$ | **!=** | $=$ | $\leq$ | $\geq$

⟨logical operator ⟩ ::= **and** | **or**

⟨repetitive action ⟩ ::=
do ⟨guarded action ⟩ **;** **...** **;** ⟨guarded action ⟩ **od**

⟨compound action ⟩ ::=
begin ⟨action ⟩ **;** **...** **;** ⟨action ⟩ **end**

⟨rule triggering ⟩ ::= **trigger off** ⟨triggering mode ⟩ ⟨rule call ⟩

⟨triggering mode ⟩ ::= **for_next**
| **for_current**
| **at_completion**

⟨predefined procedure call ⟩ ::=
⟨predefined procedure name ⟩
(⟨expression ⟩**,...,** ⟨expression ⟩**)**

⟨predefined procedure name ⟩ ::= ⟨identifier ⟩

⟨rule call ⟩ ::= ⟨rule name ⟩ **(** ⟨expression ⟩**,...** **,** ⟨expression ⟩**)**

⟨rule name ⟩ ::= ⟨identifier ⟩

⟨init action ⟩ ::= **init_action ;**
⟨variable declaration part ⟩ ⟨action part ⟩

A.3 Sample Audit Data Description File

```
A Audit Data Description File for SunOS 4.1.1
B sun4c
C 1993-04-15 10:10:23
D ASAX V1.0

1 5
2 long
3 long
4 au_time
5 time since epoch

1 6
2 int
3 int
4 record_type
5 record type os the audit record

1 7
2 int
3 int
4 uid
5 user id

1 8
2 int
3 int
4 pid
5 process id

1 9
2 string
3 string
4 filename
5 the file name
```

A.4 BNF Syntax of Datalog

⟨datalog program ⟩ ::= ⟨deduction rules part ⟩ ⟨RUSSEL interfaces part ⟩

⟨deduction rules part ⟩ ::= ⟨deduction rule ⟩ **...** ⟨deduction rule ⟩

⟨deduction rule ⟩ ::= ⟨head ⟩ ⟨tail ⟩

⟨head ⟩ ::= ⟨atom ⟩

⟨tail ⟩ ::= ⟨empty ⟩
    | **:-** ⟨atom ⟩ **...** ⟨atom ⟩

⟨atom ⟩ ::= ⟨predicate name ⟩ ⟨term list ⟩

⟨predicate name ⟩ ::= identifier

⟨term list ⟩ ::= ⟨empty ⟩
    | **(** ⟨term ⟩ **...** ⟨term ⟩ **)**

⟨term ⟩ ::= ⟨variable ⟩
    | ⟨constant ⟩

⟨variable ⟩ ::= variable beginning with an upper case letter

⟨constant ⟩ ::= ⟨user name ⟩
    | ⟨group name ⟩
    | ⟨pathname ⟩

⟨user name ⟩ ::= legal UNIX user name

⟨group name ⟩ ::= legal UNIX group name

⟨pathname ⟩ ::= absolute UNIX path name

⟨RUSSEL interfaces part ⟩ ::= ⟨empty ⟩
    | ⟨RUSSEL interface ⟩ **...** ⟨RUSSEL interface ⟩

⟨RUSSEL interface ⟩ ::=
    ⟨fact control ⟩ ⟨fact ⟩ ⟨RUSSEL rule control ⟩ ⟨RUSSEL rule name ⟩ ⟨argument list ⟩

⟨fact control ⟩ ::= **on_new**
    | **on_dispose**

⟨fact ⟩ ::= ⟨atom ⟩

⟨RUSSEL rule control ⟩ ::= **trigger**
| **cancel**

⟨RUSSEL rule name ⟩ ::= identifier

⟨argument list ⟩ ::= ⟨empty ⟩
| **(** ⟨term ⟩ **...** ⟨term ⟩ **)**

Appendix B

Detection Rules

B.1 Creation of set User id Scripts

Detects if a user has successfully created a setid program.

```
init_action;
   trigger off for_next create_setuid;

rule create_setuid;
begin
   if
      strToInt(HEADER_EVENT) = 4 and
      strToInt(RETURN_ERROR) = 0 and
      present ATTR_MODE and
      issetuid(strToInt(ATTR_MODE)) = 1
      -->
      begin
         print('User id ', strToInt(SUBJ_AUID));
         println(' has successfully setided file ', PATH_NAME)
      end
   fi;
   trigger off for_next create_setuid
end.
```

Figure B.1: Detecting the creation of setuid files

B.2 Creation of Device Files

Report all occurrences of a successful *mknod*(1) command. This would indicate that someone is trying to access a device through unauthorized means.

```
init_action;
   trigger off for_next mknod;

rule mknod;
var error, event: integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and (event = 7 or event = 23) /* exec(2) or execve(2) */
      and present ATTR_MODE and
      issetuid(strToInt(ATTR_MODE)) = 1
      -->
      begin
         print('User id ', strToInt(SUBJ_AUID));
         println(' has successfully setided file ', PATH_NAME)
      end
   fi;
   trigger off for_next mknod
end.
```

Figure B.2: Detecting the creation of device files

B.3 Detection of Executable/Setid Files Corruption

Very few programs should write to other setuid or executable files. This rule will warn when a setuid program creates or opens a setid file or executable program for write.

```
init_action;
   trigger off for_next setuid_writes_setuid;

rule setuid_writes_setuid;
var error, event, pid: integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and (event = 7 or event = 23) /* exec(2) or execve(2) */
      and present ATTR_MODE and
      issetuid(strToInt(ATTR_MODE)) = 1 /* setuid enabled */
      -->
      begin
         pid := strToInt(SUBJ_PID);
         trigger off for_next setuid_writes(pid, PATH_NAME)
      end
   fi;
   trigger off for_next setuid_writes_setuid
end;

rule setuid_writes(pid: integer; path: string);
var error, event, pid: integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and event = 1 /* exit(2) */
      and pid = strToInt(SUBJ_PID) and
      -->
      skip;
      error = 0 and event = 4 /* creat(2) */
      and present ATTR_MODE
      and pid = strToInt(SUBJ_PID) and /* same process id */
      isexec(strToInt(ATTR_MODE)) = 1 and /* executable file */
      issetuid(strToInt(ATTR_MODE)) = 1 /* setuid enabled */
      -->
      begin
         print('setuid program ', path);
         print(' created/opened for write the setuid ');
         println(' or executable file ', PATH_NAME)
      end;
      true
      -->
      trigger off for_next setuid_writes(pid, path)
   fi
end.
```

B.4 Detection of Write Access to Executable Files

Detect programs writing to executable files which may be an indication of viral behavior.

```
init_action;
   trigger off for_next writes_executables;

rule writes_executables;
var error, event, pid, euid:  integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and (event = 7 or event = 23) /* exec(2) or execve(2) */
      and present ATTR_MODE and
      isexec(strToInt(ATTR_MODE)) = 1 /* executable file */
      -->
      begin
         pid := strToInt(SUBJ_PID);
         euid := strToInt(SUBJ_EUID);
         trigger off for_next writes(pid, euid, PATH_NAME)
      end
   fi;
   trigger off for_next writes_executables
end;

rule writes(pid, euid:  integer; path:  string);
var error, event, pid:  integer;
begin
   error := strToInt(RETURN_ERROR);
   event := strToInt(HEADER_EVENT);
   if
      error = 0 and event = 1 /* exit(2) */
      and pid = strToInt(SUBJ_PID) /* same process id */
      -->
      skip;
      error = 0 and
      /* AUE_OPEN_W or AUE_OPEN_WC or ...  or OPEN_RWTC */
      (event = 76 or event = 77 or event = 78 or event = 79 or
       event = 80 or event = 81 or event = 82 or event = 83)
      and present ATTR_MODE
      and pid = strToInt(SUBJ_PID) and
      isexec(strToInt(ATTR_MODE)) = 1 /* executable file */
      -->
      begin
         print('setuid program ', path, ' euid ', euid);
         print(' created/opened for write the setuid ');
         println(' or executable file ', PATH_NAME)
      end;
      true
      -->
      trigger off for_next writes(pid, euid, path)
   fi
end.
```

B.5 Fact Base Update

```
rule udpt_fact_base(file, owner, group: string; event, mode, newmode: integer);
var pdir: string;
if
   event = 6 /* unlink */
   -->/* retract all facts related to file */
   begin
      pdir := getParentDir(file);
      retract(parentDir(pdir, file));
      retract(owner(owner, file));
      if isStartUp(file) = 1 --> retract(start_up(owner, file)) if;
      retract(worldWrite(file));
      retract(groupWrite(file, _));
      retract(worldRead(file));
      retract(groupRead(file, _))
   end;

   event = 4 /* creat(2) */
   -->/* assert all relevant basic facts related to file */
   begin
      pdir := getParentDir(file);
      assert(parentDir(pdir, file));
      assert(owner(owner, file));
      if isStartUp(file) = 1 --> assert(owner(owner, file)) fi;
      if isWorldWrite(mode) = 1 --> assert(worldWrite(file)) fi;
      if isGroupWrite(mode) = 1 --> assert(groupWrite(file, group)) fi;
      if isWorldRead(mode) = 1 --> assert(worldRead(file)) fi;
      if isGroupRead(mode) = 1 --> assert(groupRead(file, group)) fi
   end;

   event = 39 or event = 10 /* fchmod(2) or chmod(2) */
   -->/* update file permissions related to file */
   begin
      if isWorldWrite(newmode) = 1 --> assert(worldWrite(file)) fi;
      if isGroupWrite(newmode) = 1 --> assert(groupWrite(file, group)) fi;
      if isWorldRead(newmode) = 1 --> assert(worldRead(file)) fi;
      if isGroupRead(newmode) = 1 --> assert(groupRead(file, group)) fi
      if isWorldWrite(mode) = 0 --> retract(worldWrite(file)) fi;
      if isGroupWrite(mode) = 0 --> retract(groupWrite(file, group)) fi;
      if isWorldRead(mode) = 0 --> retract(worldRead(file)) fi;
      if isGroupRead(mode) = 0 --> retract(groupRead(file, group)) fi
   end;

   event = 42 /* rename(2) */
   -->/* equivalent to unlink(2) followed by creat(2) */
   begin
         /* retract all facts related to old file */
         ...
         /* assert all relevant facts related to the new file */
         ...
   end;

                                        ⋮

if;
```

Appendix C

Fagen

C.1 FDL BNF Syntax

⟨FDL file ⟩ ::= ⟨C declarations ⟩ ⟨source record declaration ⟩
⟨C declarations ⟩ ⟨target record declaration ⟩
⟨C declarations ⟩ ⟨initializers ⟩
⟨C declarations ⟩

⟨C declarations ⟩ ::= ⟨empty ⟩
| a C valid declarations and macros

⟨source record declaration ⟩ ::= **@@** ⟨source declaration ⟩ **@@**

⟨source declaration ⟩ ::= ⟨empty ⟩
| a C valid variable declaration

⟨target record declaration ⟩ ::= **@@** ⟨target declaration ⟩ **@@**

⟨target declaration ⟩ ::= ⟨type ⟩ ⟨identifier ⟩
⟨bound list ⟩ ⟨value specification ⟩ ⟨bound list ⟩

⟨type ⟩ ::= ⟨sign specifier ⟩ **char**
| ⟨sign specifier ⟩ ⟨size specifier ⟩ **int**
| ⟨sign specifier ⟩ **long**
| **float**
| **double**
| ⟨structure ⟩
| ⟨union ⟩

⟨sign specifier ⟩ ::= ⟨empty ⟩
| **unsigned**

⟨size specifier ⟩ ::= ⟨empty ⟩
| **short**

⟨structure ⟩ ::= **struct** ⟨tag ⟩
{
⟨declaration list ⟩ ;
}

⟨tag ⟩ ::= identifier

⟨declaration list ⟩ ::= ⟨component declaration ⟩
| ⟨component declaration ⟩ ⟨declaration list ⟩

⟨component declaration ⟩ ::= ⟨target declaration ⟩

⟨union ⟩ ::= **union** ⟨tag ⟩ **switch** **(** ⟨discriminant ⟩ **)**
{
⟨arm declaration list ⟩
⟨default declaration ⟩
}

⟨discriminant ⟩ ::= ⟨expression ⟩

⟨arm declaration list ⟩ ::= ⟨arm declaration ⟩
| ⟨arm declaration ⟩ ⟨arm declaration list ⟩

⟨arm declaration ⟩ ::=
case ⟨constExpr ⟩ **:** ⟨target declaration ⟩ ;

⟨default declaration ⟩ ::= ⟨empty ⟩
| **default** **:** ⟨target declaration ⟩ ;

⟨bound list ⟩ ::= ⟨empty ⟩
| [⟨size ⟩ ⟨max size ⟩] ⟨bound list ⟩

⟨size ⟩ ::= ⟨empty ⟩
| ⟨intExpr ⟩ **:**

⟨max size ⟩ ::= ⟨constExpr ⟩

⟨value specification ⟩ ::= ⟨empty ⟩
| **=** ⟨expression ⟩
| **:=** ⟨function name ⟩
| **::=** ⟨procedure name ⟩

⟨expression ⟩ ::= a C valid expression

⟨function name ⟩ ::= a C valid function name

⟨procedure name ⟩ ::= a C valid function name

⟨initializers ⟩ ::=
  ⟨start function ⟩ ⟨step function ⟩ ⟨end function ⟩

⟨start function ⟩ ::= ⟨empty ⟩
  | **at start** ⟨routine call ⟩

⟨step function ⟩ ::= ⟨empty ⟩
  | **to step** ⟨routine call ⟩

⟨start function ⟩ ::= ⟨empty ⟩
  | **at end** ⟨routine call ⟩

⟨routine call ⟩ ::= a C valid routine call

⟨intExpr ⟩ ::= a C valid expression of integral type

⟨constExpr ⟩ ::= a C valid integral constant

⟨empty ⟩ ::=

C.2 A Full Example of Using *fagen*

utmpx and wtmpx hold user and accounting information for commands such as who, write, and login. The include file describing the format of these files under Solaris 2.5 is `/usr/include/utmpx.h`. The format description file is identical to the include file except that we added the target description. In the latter, the declaration of the array `ut_host[ ]` (the remote host name) specifies the actual length of the array which is given by the field `ut_syslen`.

```
#ifndef _UTMPX_H
#define _UTMPX_H

#pragma ident "@(#)utmpx.h   1.10     94/06/02 SMI"   /* SVr4.0 1.4 */

#include <sys/types.h>
#include <sys/time.h>
#include <utmp.h>

#ifdef __cplusplus
extern "C" {
#endif

#define UTMPX_FILE      "/var/adm/utmpx"
#define WTMPX_FILE      "/var/adm/wtmpx"

#define ut_name ut_user
#define ut_xtime ut_tv.tv_sec

union utmpx {
        char    ut_user[32]; /* user login name */
        char    ut_id[4];               /* inittab id */
```

```
        char    ut_line[32];            /* device name (console, lnxx) */
        pid_t   ut_pid;                 /* process id */
        short   ut_type;                /* type of entry */
        struct exit_status ut_exit;     /* process termination/exit status */
        struct timeval ut_tv;           /* time entry was made */
        long    ut_session;             /* session ID, used for windowing */
        long    pad[5];                 /* reserved for future use */
        short   ut_syslen;              /* significant length of ut_host */
                                        /*   including terminating null */
        char ut_host[257];           /* remote host name */
};
@@
union utmpx rec;
@@
@@
struct utmpx {
        char    ut_user[32];            /* user login name */
        char    ut_id[4];               /* inittab id */
        char    ut_line[32];            /* device name (console, lnxx) */
        pid_t   ut_pid;                 /* process id */
        short   ut_type;                /* type of entry */
        struct exit_status ut_exit;     /* process termination/exit status */
        struct timeval ut_tv;           /* time entry was made */
        long    ut_session;             /* session ID, used for windowing */
        long    pad[5];                 /* reserved for future use */
        short   ut_syslen;              /* significant length of ut_host */
                                        /*   including terminating null */
        char ut_host[ut_syslen:257]; /* remote host name */
} rec;
@@


/* Definitions for ut_type */

#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5       /* Process spawned by "init" */
#define LOGIN_PROCESS 6       /* A "getty" process waiting for login */
#define USER_PROCESS    7       /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9

#define UTMAXTYPE      ACCOUNTING     /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when     */
/*      accounting for something other than a process.                  */
/*      No string for the ut_line field can be more than 11 chars +     */
/*      a NULL in length.                                               */

#define RUNLVL_MSG      "run-level %c"
```

```
#define BOOT_MSG        "system boot"
#define OTIME_MSG       "old time"
#define NTIME_MSG       "new time"
#define MOD_WIN         10

/*      Define and macro for determing if a normal user wrote the entry */
/*      and marking the utmpx entry as a normal user */
#define NONROOT_USRX    2
#define nonuserx(utx)   ((utx).ut_exit.e_exit == NONROOT_USRX ? 1 : 0)
#define setuserx(utx)   ((utx).ut_exit.e_exit = NONROOT_USRX)

#if defined(__STDC__)
extern void endutxent(void);
extern struct utmpx *getutxent(void);
extern struct utmpx *getutxid(const struct utmpx *);
extern struct utmpx *getutxline(const struct utmpx *);
extern struct utmpx *pututxline(const struct utmpx *);
extern void setutxent(void);
extern int utmpxname(const char *);
extern struct utmpx *makeutx(const struct utmpx *);
extern struct utmpx *modutx(const struct utmpx *);
extern void getutmp(const struct utmpx *, struct utmp *);
extern void getutmpx(const struct utmp *, struct utmpx *);
extern void updwtmp(const char *, struct utmp *);
extern void updwtmpx(const char *, struct utmpx *);
#else
extern void endutxent();
extern struct utmpx *getutxent();
extern struct utmpx *getutxid();
extern struct utmpx *getutxline();
extern struct utmpx *pututxline();
extern void setutxent();
extern int utmpxname();
extern struct utmpx *makeutx();
extern struct utmpx *modutx();
extern void getutmp();
extern void getutmpx();
extern void updwtmp();
extern void updwtmpx();
#endif

#ifdef __cplusplus
}
#endif

#endif /* _UTMPX_H */
```

The execution of the format adaptor generator with this FDL file as argument generates the following top-level routine for converting a native utmpx record into NADF format.

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include "vio.h"
#include "utmpx.fdf"
#include "stdlib.h"
#include "gnfalib.h"

nadfconv(void *prec, int buflen, char *buf)
{
    int N = 1;                     /* audit data id */
    char *alpha = buf + sizeof(int);/* current position in NADF buffer */
    u_short *IDL;
    struct utmpx rec;                           /* native record */
    int i1;

    rec = *(struct utmpx *) prec;
    putCharMulti(&alpha, N++, 32, (char *) &rec.ut_user, &buflen, buf);
    putCharMulti(&alpha, N++, 4, (char *) &rec.ut_id, &buflen, buf);
    putCharMulti(&alpha, N++, 32, (char *) &rec.ut_line, &buflen, buf);
    put_long(&alpha, N++, rec.ut_pid, &buflen, buf);
    put_short(&alpha, N++, rec.ut_type, &buflen, buf);
    put_short(&alpha, N++, rec.ut_exit.e_termination, &buflen, buf);
    put_short(&alpha, N++, rec.ut_exit.e_exit, &buflen, buf);
    put_long(&alpha, N++, rec.ut_tv.tv_sec, &buflen, buf);
    put_long(&alpha, N++, rec.ut_tv.tv_usec, &buflen, buf);
    put_long(&alpha, N++, rec.ut_session, &buflen, buf);
    for (i1 = 0; i1 < 5; i1++) {
        put_long(&alpha, N++, rec.pad[i1], &buflen, buf);
    }
    put_short(&alpha, N++, rec.ut_syslen, &buflen, buf);
    putCharMulti(&alpha, N++, rec.ut_syslen, (char *) &rec.ut_host, &buflen, buf);
    *((int *) buf) = alpha - buf;
}

#include <stdio.h>
#include <utmpx.h>
#include "vio.h"

#define  BUFLEN 1024
static int fd;
static void *prec;

UOPEN(int ac, char **av)
{

        /* open native file */

        if (av[0] != 0)
                utmpxname(av[0]);
```

```
        setutxent(); /* start from beginning */

        prec = (void *) malloc(BUFLEN); /* native record buffer */
        if (!prec)
                return VBADOPEN;
        return OK;
 }

int
UREAD(char **rec)
{
        struct utmpx *rc, *getutxent();

        if ((rc = getutxent()) == 0)
                return VEOF;
        *rec = (char *) rc;
        return OK;
}

int
UCLOSE()
{
        return OK;
}

main(int argc, char **argv)
{
        int rc;
        int fdout;
        char *buf;

        if (argc != 3) {
                fprintf(stderr, "%s: incorrect argument count\n", *argv);
                exit(1);
        }
        if ((fdout = creat_NADF(argv[1])) < 0) {
                fprintf(stderr, "%s: cannot create: %s: %d\n",
                        *argv, argv[1], fdout);
                exit(1);
        }
        if ((rc = vopen(argc - 2, argv + 2)) != 0) {
                fprintf(stderr, "%s: cannot open virtual stream %s\n", *argv);
                exit(1);
        }
        while ((rc = vread(&buf)) == OK) {
                if ((rc = write_NADF(fdout, buf, 1)) != 0) {
                        fprintf(stderr, "%s: cannot write: %d\n", *argv,  rc);
```

```
                        exit(1);
                }
        }
        if (rc != VEOF) {
                fprintf(stderr, "%s: cannot read virtual stream: %d\n",
                                *argv, rc);
                exit(1);
        }
        if ((rc = close_NADF(fdout)) != 0) {
                fprintf(stderr, "%s: cannot close nadf file %s\n",
                                *argv, argv[2]);
                exit(1);
        }
        if ((rc = vclose()) != OK) {
                fprintf(stderr, "%s: cannot close virtual stream\n", *argv);
                exit(1);
        }
}

putCharMulti(char **s, u_short id, int nb, char *from, int *bufLen, char **buf)
{

        u_short *ID = (u_short *) *s;
        int lg = IDLGSZ + nb*sizeof(char);
        char *to = *s + IDLGSZ;
        int dist = *s - *buf;

        if (lg%2 != 0)
                lg++;
        if (lg >= *bufLen - dist) {
                do *bufLen += CHUNK;
                while (lg >= *bufLen - dist);
                if ((*buf = (char *) realloc(*buf, (size_t) *bufLen)) == 0 ) {
                        fprintf(stderr, "couldn't realloc NADF buffer\n");
                        exit(1);
                }
                *s = *buf + dist;
                ID = (u_short *) *s;
                to = *s + IDLGSZ;
        }
        ID[0] = id;
        ID[1] = nb * sizeof(char);
        memcpy(to, from, nb*sizeof(char));
        *s += lg;
}
```

```
put_short(char **s, u_short id, short v, int *bufLen, char **buf)
{
        u_short *IDL = (u_short *) *s;
        int lg = IDLGSZ + sizeof(short);
        int dist = *s - *buf;

        if (lg >= *bufLen - dist) {
                do
                        *bufLen += CHUNK;
                while (lg >= *bufLen - dist);
                if ((*buf = (char *) realloc(*buf, (size_t) *bufLen)) == 0 ) {
                        fprintf(stderr, "couldn't realloc NADF buffer\n");
                        exit(1);
                }
                *s = *buf + dist;
                IDL = (u_short *) *s;
        }
        IDL[0] = id;
        IDL[1] = sizeof(short);
        *((short *) &IDL[2]) = v;
        *s += lg;
}

put_long(char **s, u_short id, long v, int *bufLen, char **buf)
{
        u_short *IDL = (u_short *) *s;
        int lg = IDLGSZ + sizeof(long);
        int dist = *s - *buf;

        if (lg >= *bufLen - dist) {
                do {
                        *bufLen += CHUNK;
                } while (lg >= *bufLen - dist);
                if ((*buf = (char *) realloc(*buf, (size_t) *bufLen)) == 0 ) {
                        fprintf(stderr, "couldn't realloc NADF buffer\n");
                        exit(1);
                }
                *s = *buf + dist;
                IDL = (u_short *) *s;
        }
        IDL[0] = id;
        IDL[1] = sizeof(long);
        memcpy((char *) &IDL[2], &v, sizeof(long));
        *s += lg;
}
```

```
put_double(char **s, u_short id, double v, int *bufLen, char **buf)
{
        u_short *IDL = (u_short *) *s;
        int lg = 2 * sizeof(u_short) + sizeof(double);
        int dist = *s - *buf;

        if (lg >= *bufLen - dist) {
                do {
                        *bufLen += CHUNK;
                } while (lg >= *bufLen - dist);
                if ((*buf = (char *) realloc(*buf, (size_t) *bufLen)) == 0 ) {
                        fprintf(stderr, "couldn't realloc NADF buffer\n");
                        exit(1);
                }
                *s = *buf + dist;
                IDL = (u_short *) *s;
        }
        IDL[0] = id;
        IDL[1] = sizeof(double);
        memcpy((char *) &IDL[2], &v, sizeof(float));
        *s += lg;
}
```

The generated audit data description file is

```
A sun4m
B SunOS Generic
C pascal
D Wed Mar 20 12:30:13 WET 1996


1 1
2 char
3 char
4 ut_user
5 comment

1 2
2 char
3 char
4 ut_id
5 comment

1 3
2 char
3 char
```

```
4 ut_line
5 comment

1 4
2 long
3 long
4 ut_pid
5 comment

1 5
2 short
3 short
4 ut_type
5 comment

1 6
2 short
3 short
4 e_termination
5 comment

1 7
2 short
3 short
4 e_exit
5 comment

1 8
2 long
3 long
4 tv_sec
5 comment

1 9
2 long
3 long
4 tv_usec
5 comment

1 10
2 long
3 long
4 ut_session
5 comment

1 11
2 long
```

```
3 long
4 pad_0
5 comment

1 12
2 long
3 long
4 pad_1
5 comment

1 13
2 long
3 long
4 pad_2
5 comment

1 14
2 long
3 long
4 pad_3
5 comment

1 15
2 long
3 long
4 pad_4
5 comment

1 16
2 short
3 short
4 ut_syslen
5 comment

1 17
2 char
3 char
4 ut_host
5 comment
```

Bibliography

[1] Stephen H. Brackin. An interface Specification Language for automatically analyzing cryptographic protocols. In Internet Society, editor, *Symposium on Network and Distributed System Security*, pages 40–51, 1997.

[2] J. Moore. Protocol failure in crypto systems. *Proceedings of the IEEE*, 76(5):594–602, May 1988.

[3] Dorothy E. Denning and G. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533–536, Aug 1981.

[4] G. Lowe. Some new attacks upon security protocols. In *Computer Security Foundations Workshop IX*, County Kerry, Ireland, Jun 1996.

[5] Michael B. Greenwald, Sandeep K. Singhal, Jonathan R. Stone, and David R. Cheriton. Designing an academic firewall: Policy, practice, and experience with surf. In Internet Society, editor, *Symposium on Network and Distributed System Security*, San Diego, CA, Feb 1996.

[6] T.F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *IEEE Symposium on Security and Privacy*, 1988.

[7] Teresa F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *11th National Computer Security Conference*, Oct 1988.

[8] D.E. Denning. An intrusion-detection model. *IEEE transaction on Software Engineering*, 13(2), 1987.

[9] H.S. Vaccaro and G.E. Liepins. Detection of anomalous computer session activity. In *IEEE Symposium on Security and Privacy*, May 1989.

[10] J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[11] H.S. Javitz, A. Valdes, T.F. Lunt, A. Tamaru, M. Tyson, and J. Lowrance. Next generation intrusion detection expert system (NIDES). Technical Report A016-Rationales, SRI, 1993.

[12] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H.S. Javitz, A. Valdes, and P.G. Neumann. A real-time intrusion-detection expert system. Technical report, SRI International, June 1990.

[13] U.S. Dep. Defense DoD. Dep. defense trusted computer system evaluation criteria (the orange book). Technical Report 5200.28-STD, DoD, 1985.

[14] Y. Kodratoff and R. Michalski. *Machine Learning, An Artificial Intelligence Approach, Volume III.* Morgan Kaufmann Publishers, Inc., 1990.

[15] Kevin L. Fox, Ronda R. Henning, Jonathan H. Reed, and Richard P. Simonian. A neural network approach towards intrusion detection. Harris Corporation technical Report, 1990.

[16] Hervé Debar, Monique Becker, and Didier Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium of Research in Computer Security and Privacy*, pages 240–250, May 1992.

[17] C.R Gent and C.P Sheppard. Predicting time series by a fully connected neural network trained by back propagation. *Computing and Control Engineering Journal*, May 1992.

[18] Hervé Debar. *Application des réseaux de neurones à la détection d'intrusions sur les systèmes informatiques.* PhD thesis, Université de Paris 6, 1993.

[19] H. S. Teng, K. Chen, and S. C. Lu. Security Audit Trail Analysis Using inductively generated predictive rules. In *the 6th Conference on Artificial Intelligence Applications*, pages 24–29, Santa Barbara, CA, May 1990.

[20] H. S. Teng, K. Chen, and S. C. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *the IEEE Symposium on Research in Security and Privacy*, pages 278–284, Oakland, CA, May 1990.

[21] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN 47907, August 1995.

[22] Jeremy Frank. Artificial Intelligence and Intrusion Detection: Current and Future Directions. In *17th National Computer Security Conference*, pages 22–33, Oct 1987.

[23] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Inc., 1988.

[24] S.E. Smaha. Haystack: An intrusion detection system. In *IEEE Fouth Aerospace Computer Security Applications Conference*, Orlando, FL, dec 1988.

[25] L. T. Heberlein B. Mukherjee and K. N. Levitt. Network intrusion detection. *IEEE Network*, May 1994.

[26] M.M. Sebring, E. Shellhouse, M.E. Hanna, and R.A. Whitehurst. Expert system in intrusion detection: A case study. In *11th National Computer Security Conference*, pages 74–81, 1988.

[27] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, and C. Jalali. Ides: A progress report. In *6th Annual Computer Security Applications Conference*, Dec 1990.

[28] Teresa Lunt. Ides: An intelligent system for detecting intruders. In *Computer Security, Threats and Countermeasures*, November 1990.

[29] A. Baur and W. Weiss. Audit Trail analysis Tool for systems with high demands regarding security and access control. Technical report, ZFE F2 SOF 42, Siemens Nixdorf Software, Munchen, Nov 1988.

[30] Phillip A. Porras. A state transition analysis tool for intrusion detection. Master's thesis, University of California, Santa Barbara, 1992.

[31] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Compagny, 1979.

[32] K. Ilgun. Ustat : A real-time intrusion detection system for UNIX. In *IEEE Symposium on Security and Privacy*, pages 16–29, 1993.

[33] M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. Idiot - users guide. Technical Report TR-96-050, Purdue University, COAST Laboratory, Sep 1996.

[34] S. Garfinkel and G. Spafford. *Practical Unix Security.* Computer Security. O'Reilly & Associates, Inc, first edition, 1991.

[35] Cert Advisories. Available by anonymous ftp from cert.sei.cmu.edu:/pub/cert_advisories.

[36] Matthew Bishop. Security Problems with the UNIX Operating System. Technical report, Department of Computer Sciences, Purdue University, jan 1983. Confidential Technical Memo.

[37] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *14th National Computer Security Conference*, October 1991.

[38] B. Le Charlier, A. Mounji, and M Swimmer. Dynamique Detection and Classification of Computer Viruses Using General Behaviour Patterns. In *Proceedings of the Virus Bulletin International Conference*, Boston, USA, September 1995. Virus Bulletin Ltd.

[39] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* Computer Science Press, 1992.

[40] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill Book Company, 1974.

[41] *SunSHIELD Basic Security Module Guide.* 2550 Garcia Avenue, Mountain View, CA 94043., August 1994.

[42] D. Farmer and E. H. Spafford. Cops Security Checker System. Technical report, Purdue University, West Lafayette, Indiana 47907-1398, September 1991.

[43] W. Baldwin Robert. Su-Kuang: Rule-based Security Checking. Technical report, MIT, Lab for Computer Science Programming Systems Research Group, May 1994.

[44] D. Zerkle and K. Levitt. NetKuang– a Multi-Host Configuration Vulnerability Checker. In *6^{th} USENIX Security Symposium*, San Jose, California, July 1996.

[45] Wietse Venema. Project Satan: Unix/Internet Security. In *Proceedings of the COMPSEC'95 conference*, London, 1995. Elsevier.

[46] Internet Scanner. Technical report, Internet Security Systems, 1995.

[47] A. Heydon. Specifying and Checking Unix Security Constraints. In *Proceedings of the 3^{rd} USENIX Security Symposium*, September 1992.

[48] Gene Kim and E.H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Systems Administration, Networking and Security Conference III (SANS)*, Washington DC, 1994.

[49] M. J. Maher and Raghu Ramakrishnam. Déjà vu in Fixpoints of Logic Programs. Technical report, Computer Sciences Department, University of Wisconsin-Madison, November 1989.

[50] Divesh Srivastava Raghu Ramakrishnam and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. Technical report, Computer Sciences Department, University of Wisconsin-Madison, November 1989.

[51] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *SunSHIELD Basic Security Module Guide*, August 1994.

[52] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *Network Programming Guide*, March 1990.

[53] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A User Guide to PVM (Parallel Virtual Machine). Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[54] Timothy G. Mattson Craig C. Douglas and Martin H. Shultz. Parallel programming systems for workstation clusters. Technical Report YALEU/DCS/TR-975, Yale University Department of Computer Science, August 1993.

[55] J.R. Winkler. Unix Prototype for Intrusion and Anomaly Detection in Secure Networks. 1990.

[56] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by tempo in berkley unix 4.3bsd. *IEEE Trans. on Software Eng.*, 15(7):847–853, Jul 1989.

[57] S.E. Smaha. Private Communication.

[58] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. A universal stub compiler. In *Proceedings of SIGCOMM 94*, October 1994.

[59] B. Le Charlier (Ed.). *Static Analysis: Proceedings of the First International Static Analysis Symposium.* Number 864 in Lecture Notes in Computer Science. Springer-Verlag, Namur, Belgium, September 1994.

[60] C. Ko, D. Frincke, T. Goan, L.T. Herberlein, K Levitt, B. Mukherjee, and C. Wee. Analysis of an algorithm for distributed recognition and accountability. In *1st ACM Conference on Computer Communication Security*, pages 154–164, nov 1991.