Facultés Universitaires
Notre-Dame de la Paix

Namur, Belgium

Institut d'Informatique

Rue Grandgagnage, 21
B-5000 Namur
BELGIUM

Distributed Audit Trail Analysis

Abdelaziz Mounji, Baudouin Le Charlier,
Denis Zampunieris, Naji Habra

RP-94-007

November1994

Phone: +32 81 72.49.66 Fax: +32 81 72.49.67 E-mail: cleroy@info.fundp.ac.be

Distributed Audit Trail Analysis*

Abdelaziz Mounji Baudouin Le Charlier Denis Zampuniéris
Naji Habra,

Institut d'Informatique,
FUNDP,
rue Grangagnage 21,
B-5000 Namur Belgium
E-mail: {amo, ble, dza, nha}@info.fundp.ac.be

15 November 1994

Abstract

An implemented system for on-line analysis of multiple distributed data streams is presented. The system is conceptually universal since it does not rely on any particular platform feature and uses format adaptors to translate data streams into its own standard format. The system is as powerful as possible (from a theoretical standpoint) but still efficient enough for on-line analysis thanks to its novel rule-based language (RUSSEL) which is specifically designed for efficient processing of sequential unstructured data streams.
In this paper, the generic concepts are applied to security audit trail analysis. The resulting system provides powerful network security monitoring and sophisticated tools for intrusion/anomaly detection. The rule-based and command languages are described as well as the distributed architecture and the implementation. Performance measurements are reported, showing the effectiveness of the approach.

1 Introduction

Auditing distributed environments is useful to understand the behavior of the software components. For instance this is useful for testing new applications: one execution trace can be analyzed to check the correctness *wrt* the requirements. In the area of real-time process control, critical hardware or software components are supervised by generating log data describing their behavior. The collection and analysis of these log files has often to be done real-time, in parallel with the audited process. This analysis can be conducted for various purposes such as investigation, recovery and prevention, production optimization, alarm and statistics reporting. In addition, correlation of results obtained at different nodes can be useful to achieve a more comprehensive view of the whole system.

Computer and network security is currently an active research area. The rising complexity of today networks leads to more elaborate patterns of attacks. Previous works for stand-alone computer security have established basic concepts and models [3, 4, 5, 7, 8] and described a few operational systems [1, 6, 9, 12, 18]. However, distributed analysis of audit trails for network security is needed because of the two following facts. First, the correlation of user actions taking place at different hosts could reveal a malicious behavior while the same actions may seem legitimate if considered at a single host level. Second, the monitoring of network security can potentially provide a more coherent and flexible enforcement of a given security policy. For instance, the security officer can set up a common security policy for all monitored hosts but choose to tighten the security measures for critical hosts such as firewalls [2] or for suspicious users.

A software architecture and a rule-based language for universal audit trail analysis were developed in the first phase of the ASAX project [10, 11, 12]. The distributed system presented here uses this rule-based language to filter audit data at each monitored host and to analyze filtered data gathered at a central host. The analysis language is exactly the same at both local and central levels. This provides a tool for a flexible and a gradual granularity control at different levels: users, hosts, subnets, domains, etc.

The rest of this paper is organized as follows. Section 2 briefly describes the system for single audit trail analysis and its rule-based language. Section 3 details the functionalities offered by the distributed system. Section 4 presents the distributed architecture. Section 5 describes the command interface of the security officer. In section 6, the implementation of the main components is outlined. Performance measurements are reported in section 7. Finally, section 8 contains the conclusion and indicates possible improvements of this work.

*To appear in the ISOC' 95 Symposium on Network and Distributed System Security.

2 Single Audit Trail Analysis

In this section, the main features of the stand alone version of ASAX for single audit trail analysis are explained. However, we only emphasize interesting functionalities. The reader is referred to [12] for a more detailed description of these functionalities[1]. A comprehensive description of ASAX is presented in [10, 11].

2.1 A motivating example

The use of the RUSSEL language for single 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 translated into a standard format (called *NADF*). The translation can be applied on-line or off-line. Hence, the description below is based on the NADF format of the audit trail 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 1),

RUSSEL keywords are noted in bold face characters, words in italic style identify fields in the current audit record, while rule parameters are noted in roman style. 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 *target_uid* of rule *count_rule* is needed to count only failed logins that are issued by the same user (*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

[1] Notice however that [12] is a preliminary description of a system under implementation. The examples in the present paper have been actually run on the implemented system

```
global v:  integer;

rule failed_login(max_times, duration:  integer);

if       event = 'login_logout'
  and au_text_4 = 'incorrect password'
  --> trigger off for_next
        count_rule(au_text_3,
                   strToInt(au_time)+duration,
                   max_times-1)
fi;

rule count_rule(target_uid:  string;
                expiration,
                count_down:  integer);

if        auid = suspect_auid
  and    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(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 1: **RUSSEL module for failed login detection on SunOS 4.1**

used to print results accumulated during the analysis such as the total number of detected sequences.

2.2 Salient features of ASAX

2.2.1 Universality

This feature means that ASAX is theoretically able to analyze arbitrary sequential files. This is achieved by translating the native file into a format called NADF (*Normalized Audit Data Format*). According to this format, a native record is abstracted to a sequence of audit data fields. All data fields are considered as untyped strings of bytes. Therefore, an audit data in the native record is converted to three fields[2]:

an identifier (a 2-bytes integer) identifies the data field among all possible data fields;

a length (a 2-bytes integer;)

a value i.e., a string of bytes.

A native record is encoded in NADF format as the sequence of encodings of each data field with a leading 4-bytes integer representing the length of the whole NADF record. Note that the NADF format is similar to the TLV (*Tag, Length, Value*) encoding used for the BER (*Basic Encoding Rules*) which is used as part of the *Abstract Syntax Notation* ASN.1 [14]. However, the TLV encoding is more complex since it supports typed primitive data values such as boolean, real, etc as well as constructor data types. Nevertheless, any data value can be represented as a string of bytes in principle. As a result, the flexibility of the NADF format allows a straightforward translation of native files and a fast processing of NADF records by the universal evaluator.

2.2.2 The RUSSEL language

RUSSEL (RUle-baSed Sequence Evaluation Language) is a novel language specifically tailored to the problem of searching arbitrary patterns of records in sequential files. The built-in mechanism of rule triggering allows a single pass analysis of the sequential file *from left to right.*

The language provides common control structures such as conditional, repetitive, and compound actions. Primitive actions include assignment, external routine call and rule triggering. A RUSSEL program simply consists of a set of rule declarations which are made of a rule name, a list of formal parameters and local variables and an action part. RUSSEL also supports modules sharing global variables and exported rule declarations.
The operational semantics of RUSSEL can be sketched as follows:

- records are analyzed sequentially. The analysis of the current record consists in executing all active rules. The execution of an active rule may trigger off new rules, raise alarms, write report messages or alter global variables, etc;

- rule triggering is a special mechanism by which a rule is made active either for the current or the next record. In general, a rule is active for the current record because a prefix of a particular sequence of audit records has been detected. (The rest of this sequence has still to be possibly found in the rest of the file.) Actual parameters in the set of active rules represent knowledge about the already found subsequence and is useful for selecting further records in the sequence;

- when all the rules active for the current record have been executed, the next record is read and the rules triggered for it in the previous step are executed in turn;

- to initialize the process, a set of so-called *init* rules are made active for the first record.

User-defined and built-in C-routines can be called from a rule body. A simple and clearly specified interface with C allows to extend the RUSSEL language with any desirable feature. This includes simulation of complex data structures, sending an alarm message to the security officer, locking an account in case of outright security violation, etc.

2.2.3 Efficiency

Is a critical requirement for the analysis of large sequential files, especially when on-line monitoring is involved. RUSSEL is efficient thanks to its operational semantics which exhibits a bottom-up approach in constructing the searched record patterns. Furthermore, optimization issues are carefully addressed in the implementation of RUSSEL: for instance, the internal code generated by the compiler ensures a fast evaluation of boolean expressions and the current record is pre-processed before evaluation by all the current rules, in order to provide a direct access to its fields.

3 Administrator Minded Functionalities

3.1 Introduction

The previous sections showed that ASAX is a universal, powerful and efficient tool for analyzing sequential files, in general, and audit trails, in particular.
In this section, the functionalities of a distributed version of ASAX are presented in the context of distributed security monitoring of networked computers. The implemented system applies to a network of SUN workstations using the C2 security feature and uses PVM (*Parallel Virtual Machine*) [15] as message passing system. However, the architecture design makes no assumption about the communication protocol, the auditing mechanism or the operating system of the involved hosts.

3.2 Single point administration

In a network of computers and in the context of security auditing, it is desirable that the security officer has control of the whole system from a single machine. The distributed on-line system must be manageable from a central point where a global knowledge about the status of the monitoring system can be maintained

[2] In fact, native files can be translated to NADF format in many different ways depending on the problem at hand. The standard method proposed here was however sufficient for the applications we have encountered so far.

and administered in a flexible fashion. Management of the monitoring system involves various tasks such as activation of distributed evaluators and auditing granularity control. 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.

3.3 The local and global analyses

Local analysis requirement corresponds to the ability of analyzing any audit trail associated to a monitored host. This is achieved by applying an appropriate RUSSEL module to a given audit trail of a given host. The analysis is considered local in the sense that analyzed audit data represents events taking place at the involved host. No assumption is otherwise made about which host is actually performing the analysis. Local analysis is also called *filtering* since at the network level, it serves as a pre-selection of relevant events. In fact, pre-selected events may correspond to any complex patterns of subject behaviors.

Audit records filtered at various nodes are communicated 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. Thus, concerted filtering and global analysis can lead to the detection of very complex patterns.

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.

3.4 Availability

This requirement means that a distributed evaluator must survive any of its slave evaluators failure and must easily be recovered in case of a failure of the master evaluator. The availability of a distributed evaluator ensures that if for some reasons a given slave is lost (broken connection, fatal error in the slave code itself, node crash, etc), the distributed analysis can still be carried on the rest of monitored hosts. On the other hand, if the master evaluator fails, the distributed analysis can be resumed from an other available host. In all cases, and especially for on-line analysis, all generated audit records must remain available for analysis (no records are lost). Distributed analysis recovery must also be done in a flexible way and require a minimum effort.

3.5 Logging control

This functionality involves control of the granularity of security events at the network, host and user levels. Typically, the security officer must be able to set up a standard granularity for most audited hosts and to require a finer granularity for a particular user or all users of a particular host. According to the single point administration requirement, this also means that logging control is carried out from the central machine without need for multiple logging to remote hosts.

4 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.

4.1 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. At the master host, a network level analysis subsequently takes place on the stream of records resulting from merging records incoming from slave machines. Both global and local analyses are performed by a slightly modified version of the analysis tool outlined in the previous section.

4.1.1 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).

4.1.2 Login controller

This process communicates with *auditd* in order to alter the granularity. It is able to change the set of preselected 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.

4.1.3 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,

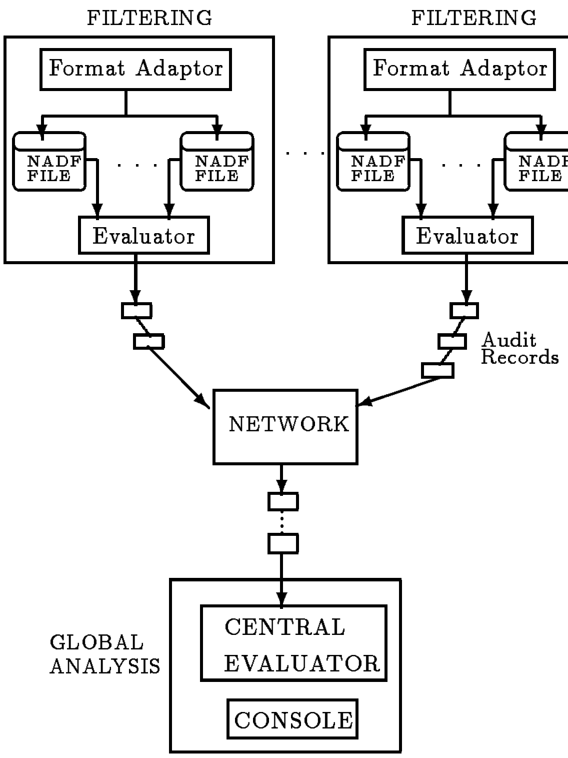

Figure 2: **System Architecture**

in the context of an heterogeneous network, they provide a standard and unique format.

4.1.4 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 6.
Audit records filtered by slave evaluators on the various monitored slave machines are sent to the central machine for global analysis.

4.2 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 2).

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

5 The Command Language of the Distributed System

5.1 Preliminaries

This section presents the command interface used by the security officer. In the following, evaluator instances are identified by their PVM 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 [16].) These audit flags are listed in Table 1.

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 1: **SunOS C2 security audit flags**

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 select 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:

5.2 Analysis control commands

The commands for distributed analysis allow to start, stop and modify a distributed analysis. 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, 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.

The **kill** command stops an evaluator identified by its instance number. **ps** reports the attributes of all active distributed evaluators. Attributes of an evaluator include instance number, instance number of the corresponding master evaluator, host name, rule module and time interval.

It is possible to activate several distributed evaluators which run independently of each others. The command **reset** stops all current distributed evaluators.

5.3 Logging control commands

The command **logcntl** implements the logging control functionality (see 3.5). It allows 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.

5.4 Example

In this section, the failed login detection example introduced in section 2.1 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 3, 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 analysis is applied. Figures 4 and 5 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

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

Figure 3: **Distributed Analysis Description File**

```
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 4: **Slave evaluator module: badlogin.asa**

this is already done by the associated slave evaluators. Figure 6 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.

6 Overview of the Implementation

The implementation of the rule-based language RUSSEL is out of the scope of this paper and is fully explained in [10, 11]. We only consider the implementation of the distributed aspects. However, it is worth noticing that very few modifications were necessary to handle record streams instead of ordinary audit trails.

In addition to the *auditd* process, the following concurrent processes are attached to each monitored host (see Figure 7):

6.1 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} is named $time_0.time_n.NADF$ where $time_0$ is the time and date found in R_0 and $time_n$ is the time stamp in R_{n-1} plus one second. 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.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

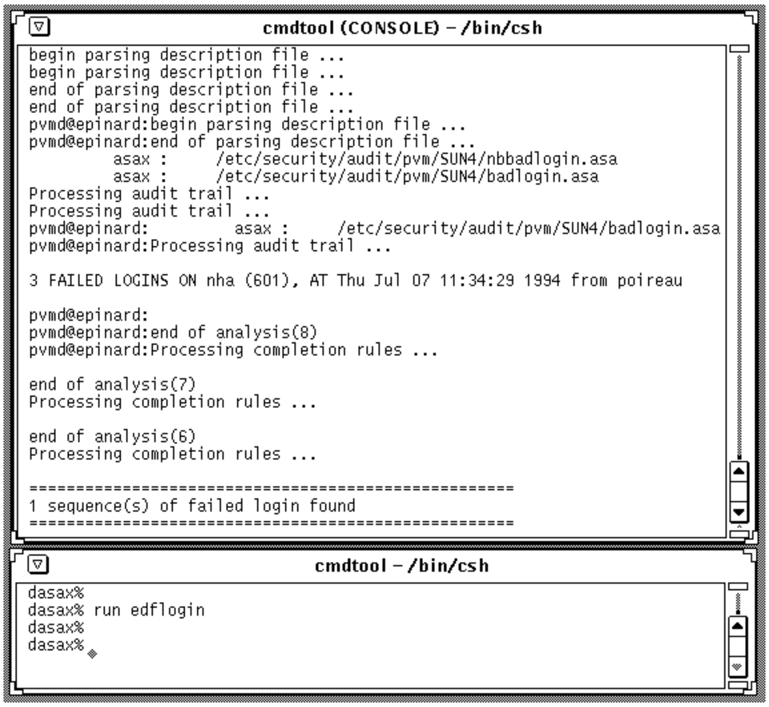

Figure 6: **Console windows**

(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`.

6.2 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 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.

6.3 Supplier process

This process runs on each monitored host. It sends to its evaluator a record stream corresponding to a given time interval. It receives from the console process on the master machine the instance number of its associated evaluator and a time interval. It retrieves corresponding records from the NADF files and sends them in sequence using a PVM message for each record. 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.

6.4 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. If the evaluator is a master evaluator, the record stream originates from a set of slave evaluators and the result of the analysis may be reports, alarms, statistics, etc. If the evaluator is a slave evaluator, there is only one sending process (the supplier process) and in this case, the result 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

```
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 5: **Master evaluator module: nbbadlogin.asa**

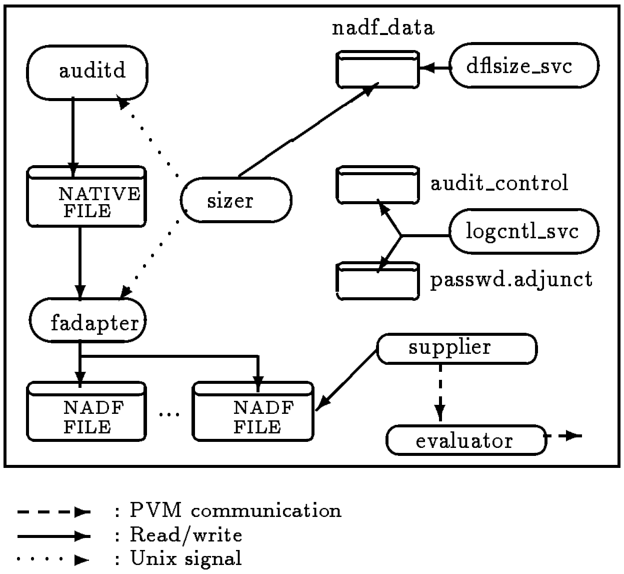

Figure 7: **Local Architecture**

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 the evaluators a 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.

6.5 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 5. 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.

7 Performance Measurements

7.1 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 due to the distributed architecture *wrt* the same analysis performed on a single audit trail. The second one compares the performance of a distributed audit trail analysis and of a centralized audit trail analysis. The last experiment shows the benefits of executing several analyses in parallel.

7.2 Overhead of the distributed architecture

In order to measure the overhead introduced by the distributed architecture, we analyzed a single audit file of 500 Kbytes using the single audit trail analysis version on the one hand and the distributed version on the other hand. The analyzed file represents a 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. In the first case, audit records are simply retrieved from the audit file using input/output routines. The second case corresponds to a degenerated distributed evaluator composed of a single slave evaluator. The overhead introduced is mainly due to network communication (using PVM) between the slave and the master. [17] describes experiments comparing 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 [17] that the average data transfer rate is around 0.049Mbytes/sec.

The slave evaluator applies the *badlogin.asa* module as explained earlier and the master evaluator runs the *nbbadlogin.asa* module. Table 2 gives the mean values of the CPU and elapsed times (in seconds) for the stand alone analysis (SAA) 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*
SAA	1.13	0.68	1.81	5.3
DA	3.43	3.73	7.20	55.7

Table 2: **Stand Alone v.s Distributed Analysis**

7.3 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[3] 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 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 machines. A system using a centralized architecture for network audit trail analysis is presented in [18].

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 3: **Distributed v.s Centralized Analysis**

7.4 Parallel v.s sequential analysis

The RUSSEL language allows 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 8.

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

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

```
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 8: **Multiple Distributed Analyses: RUSSEL modules for the master and the slaves**

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 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 9).

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.

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

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

Figure 9: **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 4: **Multiple Distributed Analysis v.s Parallel Analysis**

The distributed evaluator description file for the parallel execution is depicted in Figure 10.

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

Figure 10: **Parallel Analysis Description File**

The execution times for the parallel analysis are found in the last line of table 4. It follows from this table that the performance gain is substantial. Note that the elapsed time of the parallel 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.

8 Conclusions and Future Works

This paper 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) is specifically designed for analyzing unstructured data streams. This 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.

References

[1] A. Baur, W. Weiss, *Audit Analysis Tool for Systems with High Demands Regarding Security and Access Control.* Research Report, ZFE F2 SOF 42, Siemens Nixdorf Software, Munich, November 1988.

[2] W.R. Cheswick, S.M. Bellovin, *Firewalls and internet security: repelling the wily hacker.* Addison-Wesley 1994, 306 pages. ISBN 0-201-63357-4.

[3] D.E. Denning, *An Intrusion-Detection Model.* IEEE Transactions on Software Engineering, Vol.13 No.2, February 1987.

[4] Th. D. Garvey, T.F. Lunt, *Model-Based Intrusion Detection.* Proceedings of the 14th National Security Conference, Washington DC., October 1991.

[5] T. Lunt, J. van Horne, L. Halme, *Automated Analysis of Computer System Audit Trails.* Proceedings of the 9th DOE Computer Security Group Conference, May 1986.

[6] T. F. Lunt, R. Jagannathan, *A Prototype Real-time Intrusion Detection Expert System.* Proceedings of the 1988 IEEE Symposium on Security and Privacy, April 1988.

[7] T. F. Lunt, *Automated Audit Trail Analysis and Intrusion Detection: A Survey.* Proceedings of the 11th National Security Conference, Baltimore, MD, October 1988.

[8] T. F. Lunt, *Real Time Intrusion Detection.* Proceedings of the COMPCON spring 89', San Francisco, CA, February 1989.

[9] T. F. Lunt et. al., *A Real-Time Intrusion Detection Expert System.* Interim Progress Report, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1990.

[10] N.Habra, B. Le Charlier, A. Mounji, *Preliminary report on Advanced Security Audit Trail Analysis on Unix* 15.12.91, 34 pages.

[11] N.Habra, B. Le Charlier, A. Mounji, *Advanced Security Audit Trail Analysis on Unix. Implementation design of the NADF Evaluator* Mar 93, 62 pages.

[12] N.Habra, B. Le Charlier, I. Mathieu, A. Mounji, *ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis.* Proceedings of the Second European Symposium on Research in Computer Security (ESORICS). Toulouse, France, November 1992.

[13] A. Mounji, B. Le Charlier, D. Zampuniéris, N.Habra, *Preliminary report on Advanced Security Audit Trail Analysis on Unix* 15.12.91, 34 pages.

[14] Marshall T. Rose, *The Open Book: a Practical Perspective on OSI.* Prentice-Hall 1990, 651 pages. ISBN 0-13-643016-3.

[15] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam, *A User Guide to PVM (Parallel Virtual Machine).* ORNL/TM-11826. July, 1991, 13 pages.

[16] Sun Microsystems, *Network Programming Guide*, Part Number 800-3850-10 Revision A of 27 March, 1990.

[17] Craig C. Douglas, Timothy G. Mattson, Martin H. Shultz, *Parallel Programming Systems For Workstation Clusters.* Yale University Department of Computer Science Research Report YALEU/DCS/TR-975, August 1993, 36 pages.

[18] J.R. Winkler, *A Unix Prototype for Intrusion and Anomaly Detection in Secure Networks.* Planning Research Corporation, R&D, 1990.