Proceedings of ESORICS'92, European Symposium on Resarch in Computer Security, November 23-25 Toulouse , Springer-Verlag 1992.

ASAX: Software Architecture and Rule-Based Language for Universal Audit Trail Analysis

Naji Habra,
Baudouin Le Charlier
Abdelaziz Mounji

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

Isabelle Mathieu

Siemens-Nixdorf Software S.A.
11, rue de Neverlée
B-5080 Rhisnes
Belgium
email : isa@swn.sni.be

Abstract. After a brief survey of the problems related to audit trail analysis and of some approaches to deal with them, the paper outlines the project ASAX which aims at providing an advanced tool to support such analysis. One key feature of ASAX is its elegant architecture build on top of a universal analysis tool allowing any audit trail to be analysed after a straight format adaptation. Another key feature of the project ASAX is the language RUSSEL used to express queries on audit trails. RUSSEL is a rule-based language which is tailor-made for the analysis of sequential files in one and only one pass. The conception of RUSSEL makes a good compromise with respect to the needed efficiency on the one hand and to the suitable declarative look on the other hand. The language is illustrated by examples of rules for the detection of some representative classical security breaches.

1 Introduction

An ideal level of security[1] could be attained by a computer system if the concerned operating system together with its related networking software and hardware utilities

[1] quoted level "A" in the US Evaluation Criteria [TCSEC]; level "G7" in the German National Criteria [GISA-EC]; level "E6" in the standardized European Information Technology Security Evaluation Criteria [ITESC].

can be formally proven to be completely secure against all attacks, e.g., intrusions, viruses,... Obviously, actual systems are far from this formal rigour. Operating systems and communication protocols are very complex devices including heterogeneous components; their theoretical basis involve temporal aspects that make their formalization more complex. In addition, there is no exhaustive model for all kind of attacks, merely because all the possible attacks are not known in advance. And, as long as such a high level of rigour is not attained, an important security task should be achieved by the observation of the system at work. Generally, operating systems generate a description of some selected aspects of their behaviour in a so-called audit trail. Thus, intelligent analysis of audit trails constitutes an important task of security management.

This paper outlines the project ASAX, "Advanced Security Audit-trail Analysis on uniX" which aims at supporting intelligent analysis of audit trails. In particular it describes a general rule-based language for such an analysis, viz. RUSSEL "RUle-baSed Sequence Evaluation Language". The paper is organized as follows: Section 2 highlights the main problems related to audit trail analysis and examines some existent approaches to deal with them. Section 3 describes the ASAX project and Section 4 illustrates the language utilization by some examples. Finally, Section 5 presents some concluding remarks.

2 Audit Trail Analysis: Problems and Approaches

Audit trail analysis should be achieved in an efficient and intelligent way. The design of an automated tool for such an analysis sets some particular problems related to:

- the disparity of security break scenarios,
- the huge amount of data generally involved by audit trails,
- the reusability of audit trails and of the analysis tools, and
- the conviviality of the interface supplied to the security manager.

These problems and the different approaches to deal with them are discussed below.

2.1 Disparity of security breaches

Survey works, e.g., [Anderson80], [Lunt88b], distinguish different kinds of security breaches that can be addressed by audit trail analysis. To each kind of attacks corresponds some characteristic symptoms detectable (at least theoretically) by an adequate analysis. Let us give a classification of those attacks together with some representative symptoms.

i- *External Penetrations*. Penetrations by users who are not authorized to use the system. A characteristic symptom in this case could be, for example, the repetition of failed login commands.

ii- *Internal Penetrations*. Penetrations by users who are authorized to use the system; these include:

  ii-a *Authorized users* who attempt to go beyond their access privileges to some resource. The symptom in this case could be the repetition of commands which are failed for denied access and/or commands which change access mode and ownership.

ii-b *Masqueraders* who operate under another user identity. The detection of such penetrations requires an adequate description of users' normal activities: the so-called "profiles" of users. A significant deviation from such profiles could then be used as a masquerader penetration symptom.

ii-c *Misusers* who are authorized to use the system and have the access privileges but who misuse their rights. Such a security breach are difficult to detect by audit trail analysis. The symptoms of cases ii-a and/or ii-b could be used. Another kind of security measures is still necessarily to minimize misuses by administrators, e.g., separating operator and administrator functions, separating system administration role from security administration role,…

iii- *Viruses and Trojan horses.* A known virus, could be characterized by a succession of events (a scenario). An example of scenario is "getting the date of an executable file, reading the file, rewriting the file which has been grown meanwhile then restoring the date". On the other hand, detecting unknown viruses requires a sufficiently large and detailed description of system's normal activities, the so-called system profile, which describes the mean of CPU, the average number of read/write operations related to each executable file,… A significant deviation from such "normal behaviour" could then be used as virus symptom.

The above symptoms suggests to classify the approaches of audit trail analysis, roughly speaking, in two families: i/ approaches based on a statistical modelling of an average "normal activity" of users, processes, the whole system,... and ii/ approaches based on a modelling of experts' knowledge about "predictable penetration". The former approaches are appropriate to detect known penetration scenarios and the latter ones are appropriate to detect unknown scenarios.

The adequacy and feasibility of statistical approaches have been proven by the early Systek work [Lunt86] and the first versions of the IDES system [Lunt88a]. On the other hand, virus detection systems, e.g., [Brunnstein91], are based on a priori knowledge about known viruses and viruses in general. The Midas system [Whitehurst87] and the more recent IDES versions [Lunt90] integrate statistical knowledge about what could be considered as normal behaviour together with heuristics knowledge about what could be considered as suspected one, in a single knowledge base. The audit trail analysis tool becomes then an expert system whose role is not only to detect security breaches but also to enhance its own knowledge, e.g., updating the profile of some user, incorporating the scenario of some new attack,...

2.2 Amount of data and selection problem

An important factor that makes audit trail analysis difficult is related to the size of audit trails. Actually, in the absence of adequate selection, operating systems produce mountains of audit data that make the analysis impossible in practice. And, even if the operating system allows auditing to be selective at the source, the problem remains that security officer must know exactly what he/she looks for. Inadequate selection may also produce unusable audit trails.

Intelligent auditing must then provide adequate and appropriate selection. Such a selection can be achieved either at the source level, in this case it is called *preselection*, or at the analysis level, it is then called *postselection*. In practice, a preprocessing filtering phase, that makes use of some simple rules, could relieve the analysis tool of a lot of irrelevant data. This is the case in the systems Midas and IDES [Whitehurst87][Lunt89a,89b,90].

Another problem related to the amount of data to be analyzed lies on the choice between on-line analysis and off-line analysis. In their experimental phase, systems generally achieve off-line analysis, but the ultimate goal is often to have an on-line analysis performed using dedicated machine resources.

2.3 Reusability: Genericity and Universality

A worthy quality of a software is its ability to be reused on different environments with a minimum of adaptation effort. In software engineering recent literature, the quality of *reusability* is addressed at the different levels of the software development. This includes reusability of code, detailed design, global design or even of specification components. Obviously, audit trail analysis tools are not excepted from 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. In addition, the auditing tool in one given organization may be expected to analyze different kinds of audit trails produced by different machines in a heterogeneous net.

Reusability of auditing tools can be considered either at the logical level or at the physical level. Reusability at the logical level is well exemplified by the IDES model underlying the IDES system. This model [Denning87] provides a general description of an intrusion detection system made by means of abstract components: subjects, objects, profiles, activity rules,… The IDES model is becoming a *de facto* standard in the domain because it provides a general framework allowing other auditing tools to be described in a general and abstract way. This model eases not only the portability of the logical design of a system but also the comparison between different systems and the expertise exchange. The ASAX project, described in Section 3, considers the reusability requirement at the physical level as well.

One way to achieve reusability is to conceive a parameterized analysis tool which can be instantiated for the different audit trail formats; in this case we talk of *generic* tool. Another way to achieve the sought reusability is to conceive a general tool allowing the analysis of any audit trail which must be previously translated to a format appropriate to that tool; in this case we prefer to talk of *universal* tool. In this sense, ASAX analysis tool is a universal one.

2.4 User Interface

An auditing system should have a suitable user interface allowing security officer to converse easily with the system and to take advantage of all its features. Practically, the purpose is to make a compromise between a powerful language allowing to express complex queries and to update the system knowledge, and an easy but less powerful language which does not require tedious training.

3 ASAX project

The project ASAX is being developed jointly by the University of Namur and Siemens Nixdorf Software S.A. The ultimate goal is to define and implement an expert system for *universal, efficient* and *powerful* audit trail analysis corresponding to security level B3. The following sections outlines the project according to three axes corresponding to those qualities.

3.1 Universality: The architecture

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 analyse any audit trail provided this trail has been previously translated to an appropriate format , ii/ to ensure the maximum of efficiency which remains the major concern in audit trail analysis.

Practically, we defined a normalized audit file format (NADF) which is sufficiently flexible that all 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 is demonstrated by providing format adaptors for existing audit trails on BS2000 and SINIX, two operating systems of Siemens.

The translation feasibility is assured by the fact that the NADF format is extremely simple and flexible. A NADF file is actually a chronological sequence of records; a record consists of the record real length and a list of audit data each represented by an identifier, the data length and the data value; the type of the data value is uniquely determined by the identifier.

In addition, format translator generates other auxiliary files that conserve the connection between external and internal formats of audit data: association tables, decoding rules… This allows analysis queries to be expressed using the external format. The architecture is sketched in Figure 1 below.

In subsequent phases of the project, the idea of providing files to encapsulate the internal to external format correspondence could be generalized. For example, providing a higher level logical description of the audit trail records, (e.g., in terms of an IDES-like model objects and subjects) allows queries to be expressed in a higher level language (e.g., in terms of the concerned model subjects and objects). In Figure 1, the extension is depicted in dashed lines.

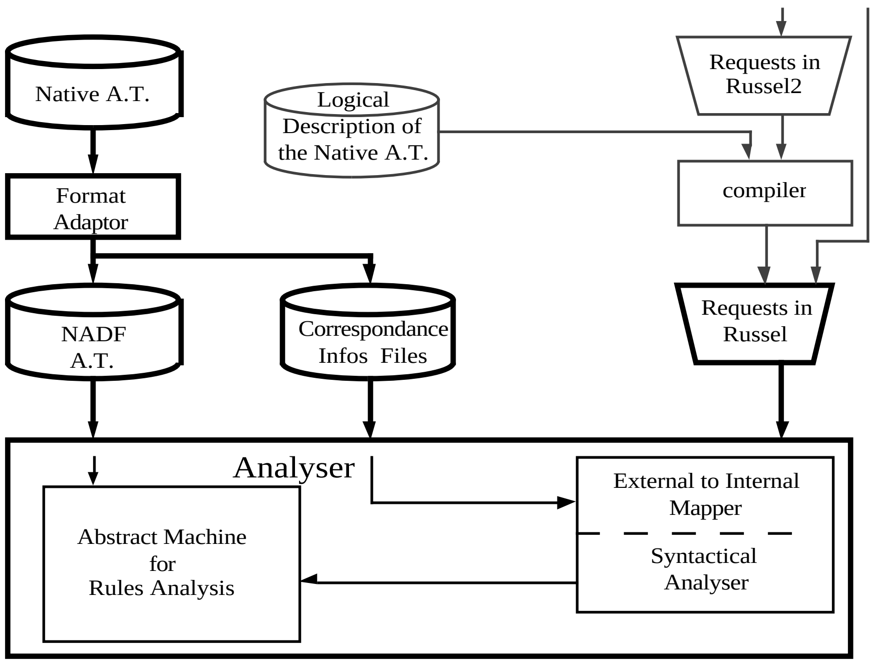
Native A.T.
Logical Description of the Native A.T.
Requests in Russel2
compiler
Format Adaptor
NADF A.T.
Correspondance Infos Files
Requests in Russel
Analyser
Abstract Machine for Rules Analysis
External to Internal Mapper
Syntactical Analyser

Figure 1

The simple format of NADF guarantees the portability at the physical level. The experience with BS2000 and SINIX shows that writing format adaptors is quite straightforward. Some complex audit records (e.g. those having repetitive items) are merely split into sequences of NADF records.

3.2 Power : The 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 specially tailored for processing large sequential files efficiently. The power of rule-based languages is generally recognized: the intrusion detection system IDES [Lunt90] uses a general rule-based expert-system tool PBEST, the expert system for virus detection of the University of Hamburg [Brunnstein91] uses rules a la OPS-5 while the OSIRIS system [Baur88] uses Prolog rules.

However in audit-trail analysis, a general-purpose rule-based language should not necessarily allow encoding any kind of declarative knowledge or making a general reasoning about that knowledge. For such purpose, a rule-based language will be used in a well established way: recognizing particular patterns in sequential files and triggering off appropriate actions, e.g. activating alarms, sending messages. The idea of RUSSEL is to take advantage of such particular utilization to make the language as efficient and easy to use as possible.

Thus, RUSSEL could be viewed as a procedural language including a particular *predefined control structure* which is suitable to make reasoning about sequences of records. This control structure is *based on a rule triggering mechanism*, it could be roughly summarized as follows:

- The audit trail is analyzed sequentially record by record by means of a collection of rules. At a given time, there is a current record being analyzed and a collection of rules which are active.
- Active rules encapsulate all the relevant knowledge about the past of analysis. This knowledge is then applied to the current record by executing the rules for that record. The process, in turn, generates new rules to be applied latter.
- From the programmer point of view, a rule is like a classical parameterized procedure but which may involve statements for a particular kind of actions: triggering rules off.
- Triggering a given rule off involves supplying the actual parameter of the rule and specifying when the rule is to be triggered off: for the current record, the next record or at the end of analysis.
- The process is initialized by a set of rules activated for the first record.

Therefore, programming a query for analyzing an audit trail in RUSSEL consists in writing a number of adequate rules which will be activated according to the global control scheme above. The fact that audit trails are chronologically sorted, obviously, plays an essential role in the approach: advanced audit trail analysis consists mainly in searching for complex chronological subsequences in the stream of all the recorded events.

A rule *declaration* in RUSSEL is made of a name, a set of formal parameters, a set of local variables and an action part. An *active* rule is the instantiation of a rule declaration with actual parameters. Active rules are executed once at a time (there is only *one* currently executed rule). Classical control structures can be used to write conditional actions, repetitive actions, sequential actions, and so on. Elementary actions include assignment, rule triggering and external routines call. The latter action allows programmer to use procedures written in an external language, like C, for reading input, writing output, accessing complex data structure and so on; external routines can be called to achieve any feature which has nothing to do with the sequential processing of the audit trail.

Section 4 includes some sample rules. One can observe the essential use of conditional actions to make the rules selective. In fact, actions are generally to be applied only on some audit records (e.g., those corresponding to a given event, concerning a given subject,…).

It should also be noted that a rule is to be re-triggered explicitly if it should survive for the analysis of the next record, by default a rule expires after its triggering. The idea is to have a simple and direct control on the rules life and to avoid complex mechanisms for killing obsolete rules.

The presentation of the language RUSSEL above, together with the examples in section 4, shows that RUSSEL is powerful enough to support any query about sequential audit trail one may need. However, the language could appear too difficult to use because it involves a different style of programming. Programmer should make reasoning taking into account the control structure implied by the rules triggering mechanism. This complexity seems acceptable for two reasons:

- In comparison with less efficient general purpose rule-based languages, RUSSEL does not appear more difficult to use. For example, IDES rules which are programmed in PBEST [Lunt90] have the same expressive power as RUSSEL ones. In PBEST, programmer has also to manage rule triggering and rule killing; but this has to be done indirectly by asserting and removing facts. The reason is that PBEST intends to preserve a certain declarative view of its knowledge. In RUSSEL, a more pragmatical approach is taken: the efficiency of procedural languages is recognized and fully exploited. Nevertheless, actual rules can often be written, or read, in a fairly declarative way. So, we claim that the language provides a very good compromise with respect to efficiency and declarativeness (although it is not declarative at all in theory).

- The underlying idea of the project ASAX, is to provide a language which is powerful, portable and implementable in a very efficient way. However, RUSSEL is not to be considered as the final user language in which security officers will build their queries from scratch, but only as an intermediate language. In subsequent phases, a library of higher level predefined selection rules will be defined. Furthermore, it is planned to design a higher level language, say RUSSEL2, allowing queries to be expressed in terms of a higher level model (e.g., using objects and subjects) and more declarative style. Queries in RUSSEL2 will be automatically translated in RUSSEL. The higher level language RUSSEL2 will be easier to use than RUSSEL but less powerful; thus, some unusual queries will still be programmed directly in the intermediate language RUSSEL. The idea is sketched in Figure 1.

3.3 Efficiency : The implementation

Efficiency is a critical aspect of audit trail analysis because of the large amount of data to be analyzed. The needed efficiency is achieved by two key principles underlying ASAX design: i/ the analysis of the audit trail is done in one and only one pass and; ii/ the repetitive steps are optimized as much as possible (processing the current record by a set of active rules and shifting from the current record to the next one).

The first principle is met by the language RUSSEL which is precisely designed with the aim of allowing any query to be processed in one and only one pass. The needed information about the past of the current record is encapsulated in the set of active rules. In addition, since RUSSEL supports external routines call, global information about the past can also be stored inside C data structures and accessed via external routines.

The second principle is achieved by means of efficient implementation techniques which carry out the needed optimization. A detailed description of the implementation is out of the scope of this paper (see [Asax2] for a comprehensive description). In what follows, only some main ideas are mentioned:

- RUSSEL rules are translated into an optimized internal code which can be processed efficiently by an appropriate abstract machine. A main (although simple) optimization idea at this level consists in transforming boolean conditions that are naively represented as “and/or” trees into more appropriate binary trees in which each node includes one elementary condition to be evaluated, say Cond, and two pointers: one points at the next condition to be evaluated if Cond is true and the second points at the next condition to be evaluated if Cond is false. This way, most elementary conditions are not evaluated at all in many cases. Since rules would include mainly conditional forms: “Condition->Action”, this techniques can be very rewarding.

- Since the system will be, most of the time, evaluating rules that heavily involve the different data fields in the current record, an efficient access to the current record is critical. Remember that the current record has variable length and free format. Thus, before applying the active rules on the current record, the record is processed once in order to create an optimized indirection table containing pointers at the different data of the record. During the processing of the current record, audit data will be accessed via this indirection table. Actually, a template of the indirection table is prepared in a fixed location area once for all at the beginning and updated once each time a record becomes current. So, every reference to a data in the rule internal code could be represented by the location of the corresponding entry in the indirection table template, in such a way that all needed audit data are directly available during the current record processing. This minimizes the access time during current record processing, all references to data being absolute addresses. (A reference to a field of the current record is the address reserved for this field in the indirection table).

- The system preserves at each time three sets of rules: those active for the current record, those to be activated for the next one and those to be activated at the end of the whole audit trail processing. Each set is represented by a linked list each cell of which contains two pieces of information: a pointer to the internal code of the rule and a list of actual parameter values (corresponding to a triggering of this rule). Notice that many instances of the same rule may exist and all use the same internal code. Passing from the current record to the next one is made straightforward by shifting from the set of current rules to the set of next rules and by re-initializing the set of next rules to the empty set.

- When the processing of a rule is started, the actual parameters are copied into a field storage location which is the same for all rules. This allows direct access to the actual parameter values.

3.4 Other features of ASAX

In a further version, some extensions are also planned. For example, ASAX will provide a *parameterized* format adaptor that achieves the format adaptation starting with a declarative description of the native audit trail format. We also plan to offer the possibility of interleaving the format adaptation function with the analysis function in order to increase the efficiency. Notice that ASAX architecture does not make any a priori assumption about the choice between an off-line aor an on-line analysis.

4 Sample Rules

The examples developed hereafter show the utilisation of RUSSEL in the detection of the typical security breaches (i), (ii-a), (ii-b), and (iii), according to the classification given in Section 2.1. To each typical security breach, a simplified symptom is first assumed, a set of RUSSEL rules is then proposed to detect that symptom. The aim is not to develop very complete and sophisticated selection but only to show how RUSSEL deals with typical situations in audit trail analysis.

Notice that, RUSSEL rules admit actually parameters of simple types only; access to structured object should be done via external routines. However, we assumed that a rule could have parameters of structured types (viz., a table parameter). The aim is to make the examples more understandable.

In the following examples, external procedures calls are noted in italic, RUSSEL keywords are noted in bold face characters, “evt”, “res”, “terminal”, “userid”, and “timestp” identify fields of the audit data record. Notice that presence of the fields is always assumed to be optional (a non present field has a special conventional value)

4.1 Example A: Detection of an external penetration

The symptom to be detected is the occurrence a number “maxtime” of failed login’s during a period of “duration” seconds.

```
rule  Failed_login (maxtimes , duration : integer)

# This rule detects a first failed login and triggers off an accounting rule with an
# expiration time

begin

if  evt='login'   and   res='failure'   and   is_unsecure (terminal)
    —> Trigger off for next    Count_rule1  (maxtimes-1, timestp+duration )
fi;

Trigger  off  for  next   Failed_login ( maxtimes , duration)

end
```

```
rule  Count_rule1  ( countdown , expiration : integer)

# This rule counts the subsequent failed logins,
# it remains active until its expiration time or until the countdown becomes 0

if   evt='login'    and    res='failure'
    and  is_unsecure(terminal)    and    timestp < expiration
    —>    if      countdown >1
          —>      Trigger off for next  Count_rule1(countdown-1,  expiration) ;
                  countdown =1
          —>      SendMessage  ("too much failed login's")
          fi  ;

    timestp ≥ expiration
    —>  Skip;

    true
    —>    Trigger off for next  Count_rule1  (countdown,  expiration)

fi
```

It should be noted here that the rule is doing nothing if the time stamp *timpstp* is greater than the expiration time *expiration.* This has the effect of killing the rule (explicit retriggering is required to perpetuate it).

4.2 Example B: Detection of a penetration by an abusive user

The symptom to be detected is the occurrence of a number “maxtime” of commands failed for non sufficient access privilege during a period of “duration” seconds.

```
rule  Abusive_user        (maxtimes , duration : integer)

# This rule detects a first failed command and triggers off an accounting rule;

begin

if  (       (evt=‘chown’ and  res=‘failure’ )  or (evt=‘chmode’ and  res=‘failure’ )
       or   (evt=‘cd’  and  res=‘failure’ )         or (evt=‘open’ and  res=‘failure’) or … )
     —>
     Trigger off for next Count_rule2  (maxtimes-1,  timestp+duration,  userid)
fi  ;

Trigger  off  for  next   Abusive_user (maxtimes , duration)

end
```

```
rule        Count_rule2     (countdown , expiration , suspectid : integer)

# This rule counts the other failed commands of the suspected user,
# it remains active until its expiration time or until the countdown becomes 0

if  (       (evt=‘chown’ and  res=‘failure’ )  or (evt=‘chmode’ and  res=‘failure’ )
       or   (evt=‘cd’  and  res=‘failure’ )         or (evt=‘open’ and  res=‘failure’) or … )
     and    userid =suspectid
     and     timestp < expiration
     —>
     if     countdown>1
            —>  Trigger  off  for  next
                            Count_rule2  (countdown-1, expiration, suspectid);
            countdown=1
            —> SendMessage  (“too much access break attempts for:”, suspectid)
     fi

     timestp ≥ expiration
     —>   Skip  ;

     true
     —> Trigger off for next Count_rule2(countdown,   expiration  , suspectid)

fi
```

4.3 Example C: Detection of an internal penetration by a masquerader

The symptom to be detected is a suspected session of use, that is a login of a user issued from a terminal, at a time which are non habitual for that user followed by a use beyond the habitual use of that user. This habitual use is described by a pre-established profile. A (very simplified) profile is a table in which each entry represents a command and a maximum occurrences for that command admissible during, say, 1 hour.

```
rule Masquerader()

# This rule detects a login from a non habitual terminal, at a non habitual time;
# it then triggers off a rule to watch the concerned suspected user providing this
# rule with the corresponding profile.

begin

if  evt='login'    and    not habitual_term( userid, terminal)
                   and    not habitual_time( userid, timestp)
    —>    Trigger off for next Watch(userid,  profileof(userid)  ) ;
fi ;

Trigger  off  for  next   Masquerader()

end
```

```
rule  Watch  ( suspectid: integer ; profile:table[cmnd:byte_string, maxtimes:int] )

# This rule watches a suspected user; when it detects a first occurrence of a
# command belonging to the user's profile it triggers off an accounting rule for
# that user and that command and provides this rule with an expiration time.
# The rule remains active until a login of user with habitual terminal and time.

if  userid = suspectid  and  evt='login'
    and  habitual_time( userid, timestp) and habitual_term( userid, terminal)
    —> skip;

    userid = suspectid  and  is_an_entry_in( evt, profile)
    —>    begin  maxtimes := select(evt,profile);
                 Trigger  off  for  next
                   Count_rule3 (suspectid, maxtimes, timestp+ 3600, evt );
                 Trigger  off  for  next Watch(  suspectid,  profile)
          end  ;

    true
    —> Trigger  off  for  next  Watch(   suspectid,  profile)

fi
```

```
rule Count_rule3  (suspectid, countdown, expiration:integer ; cmnd:byte_string )

# This rule counts the occurrences of a command "cmnd" for a suspected user,
# it remains active until its expiration time or until the countdown becomes 0.

if  userid=suspectid   and   evt=cmnd   and   timestp <expiration
    —> if  countdown > 1
                 —>  Trigger off for next
                         Count_rule3 (suspectid, countdown-1, expiration, cmnd);
           countdown = 1
                 —> SendMessage  ("unusal behaviour for:" suspectid)
         fi ;

    timestp ≥ expiration
    —>  Skip  ;

    true
    —> Trigger off for next  Count_rule3  (suspectid,countdown,expiration,cmnd);

fi
```

4.4 Example D: Detection of a suspected scenario

The symptom to be detected is a suspected scenario occuring a number of times exceeding a predefined value. A (simplified) scenario is a succession of commands, it could be represented by a table in which each entry represents a command.

```
rule  Suspect_scenario  ( scenario: table[cmnd:string];  i, n, countdown : integer)

# This rule watches the progress of a suspected which is represented by a table.
# i is the current index in the table (the next command expected to occur); n is the
# table length; countdown is initialized to the maximum number of times the
# scenario could occur. The rules remains active until the countdown becomes 0.

if  evt = scenario[i]
    —>   if      i<n  —> Trigger off for next
                              Suspect_scenario(scenario,i+1,n,countdown  );

                 i=n  —>  if countdown > 1
                          —>  Trigger  off  for  next
                                Suspect_scenario( scenario, 1, n, countdown-1 ) ;
                          countdown = 1
                          —> Alarm('suspect scenario occurs')
                           fi
           fi ;

    true
    —>   Trigger off for next  Suspect_scenario ( scenario, i, n, countdown )

fi
```

5 Conclusion

The project ASAX aims at providing a universal, powerful and efficient expert system to analyze audit trails. In order to prove the feasibility of such a system and, in the same time, to follow an incremental development methodology, the first phase of the project produces a *prototype* of the target system.

Classical classification of prototyping approaches (see e.g., [Habra91]), distinguish *horizontal* prototyping where the prototype concerns only some critical aspects of the target system disregarding the other ones, and *vertical* prototyping where all the aspects are concerned but only for some components of the target system. According to such classification, our first phase represents a vertical prototyping and an horizontal one as well.

The first phase represents an horizontal prototyping. It concerns only some aspects of the final system, viz. efficiency, power and portability; other aspects like user-friendliness will be addressed in subsequent phases. In fact, we believe that the selected aspects are the most critical ones in audit-trail analysis, and thus, the feasibility of the project depends on them. Efficiency and power goal are met by the design of the language RUSSEL. This language is implemented in a very efficient way and, in the same time, it is sufficiently powerful to allow any query on audit trail to be expressed in that language and to be handled in one pass. The portability goal is met by the definition of a very simple standardized format for audit trail files. This simple format eases the construction of format adaptors for the native audit trails of two very different operating systems, viz. BS2000 and SINIX. We believe that format adaptors for other native audit trails would not present more difficulties.

The first phase represents also a vertical prototyping. It addresses some parts of the final system, viz. the rule-based language and some sample format adaptors; it postpones the other parts, e.g. a high-level query language and a user-friendly interface. In fact, the selected parts constitutes the core of the target system. In order to make the system operational and usable by an average security officer, it will be necessary to build a large library of reusable rules, to design a higher level query language and to provide a user-friendly interface.

In addition, all the technical aspects concerning the relationship between the production and the analysis of audit data should also be addressed, e.g., the choice between on-line or off-line analysis, the analysis on the audited machine (network) or on another dedicated machine,… These aspects are related to the architecture of the system to be audited, they necessitate ad-hoc solutions. ASAX does not make any a priori assumptions about these architectures.

Appendix : Abstract Syntax for RUSSEL

The abstract syntax formalism is borrowed from [Tennent81]

Syntactic domains

A	actions	O	arithmetic operators
B	logical operators	P	formal parameters
C	conditions	Q	rule names
E	expressions	R	rules
F	field names	S	relational operators
G	parameter declarations	T	types
H	variable declarations	V	local variables
L	literals	X	external function names
M	triggering modes	Y	external procedure names

```
R     ::=    rule Q ( … ; G ; …) ; … ; H ; … ; A
G     ::=    P : T
H     ::=    V : T
A     ::=           V := E                        |   Y ( … , E , … )
             |  trigger off  M  Q  ( … , E , … )  |  begin  …  ; A ;… end
             |  do   …  ;  C —> A ;  … od          |  if  … ; C —> A ; … fi
C     ::=    true  |  F present  |  not C  | C  B  C  |   E  S  E
E     ::=    L  |  V  |  F  |  P  |  -E  |  E  O  E  |  X ( … , E , … )
B     ::=    and   | or
O     ::=    +  |  -  |  *  |  div  |  mod
S     ::=    >  |  <  |  =  |  ≠  |  ≤  |  ≥
M     ::=    for  current   | for  next   | at  completion
T     ::=    integer   |   byte_string
```

References

[Anderson80] J.P. Anderson, "Computer Security Threat Monitoring and Surveillance", J.P. Anderson Co, Fort Washington, PA, April 1980.

[Asax1] N. Habra, B. Le Charlier, A. Mounji & I. Mathieu, "Preliminary Report on Advanced Security Audit Trail Analysis on UniX", Research Report 1/92, Institut d'Informatique, University of Namur, January 1992.

[Asax2] N. Habra, B. Le Charlier & A. Mounji, "Advanced Security Audit Trail Analysis on UniX: Implementation Design of the NADF Evaluator", Research Report 7/92, Institut d'Informatique, University of Namur, March 1992.

[Baur 88] 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, München, November 1988.

[Brunnstein91] K. Brunnstein, S. Fisher-Hübner & M. Swimmer, "Concepts of an Expert System for Virus Detection", Proceedings of the 7th IFIP International Conference and Exhibition on Information Security, Brighton, UK, May 1991.

[Denning 87] D.E. Denning, "An Intrusion-Detection Model", IEEE Transactions on Software Engineering, Vol.13, No.2, February 1987.

[Garvey91] Th.D. Garvey & T.F. Lunt, "Model-Based Intrusion Detection", Proceedings of the 14th National Security Conference, Washington DC., October 1991.

[Habra 91] N. Habra, "Computer-Aided Prototyping : A Transformational Approach", *Information and Software Technology,* Vol.33, No.9, November 1991.

[Lunt 86] 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.

[Lunt 88a] 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.

[Lunt 88b] T.F. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey", Proceedings of the 11th National Security Conference, Baltimore, MD, October 1988.

[Lunt 89a] T.F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst & S. Listgarten "Knowledge Based Intrusion Detection", Proceedings of the 1989 AI Systems in Government Conference, Washington, DC., March 1989.

[Lunt 89b] T.F. Lunt, "Real Time Intrusion Detection", Proceedings of the COMPCON spring 89', San Fransisco, CA, February, 1989.

[Lunt 90] 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.

[Tennent81] R.D. Tennent, "Principles of Programming Languages", Printice-Hall International, 1981

[Whitehurst87] R.A. Whitehurst, "Expert Systems in Intrusion Detection: A Case Study", Computer Science Laboratory, SRI International, Menlo Park, CA, November 1987.

Standard Manuals

[TCSEC] "Trusted Computer System Evaluation Criteria", *The Orange Book*, Department of Defense, NCSC, National Computer Security Centre, DoD 5200.28-STD, December 1985.

[GISA-EC] "Manual for the Evaluation of Trustworthiness of Information Technology Systems", The German Information Security Agency (GISA) , February 1990

[ITSEC] "Information Technology Security Evaluation Criteria", European Community Advisory Group SOG-IS, June 1991