Preliminary report on **A**dvanced **S**ecurity **A**udit Trail Analysis on uni**X** (ASAX also called SAT-X)

Naji Habra Baudouin Le Charlier
Abdelaziz Mounji,
Institut D'Informatique,
FUNDP,
rue Grangagnage 21,
5000 Namur
E-mail: {nha, ble, amo}@info.fundp.ac.be

Isabelle Mathieu,
Siemens-Nixdorf Software S.A.,
rue de Neverlée 11,
5080 Rhisnes
E-mail: isa@swn.sni.be

September 26, 1994

Contents

.

Chapter 1

Overview of the objectives

The ASAX project is a joint project involving SWN in Rhines and the Institut d'Informatique (FUNDP) in Namur. This project aims at defining and implementing a commercial system for universal, efficient and powerful audit trail analysis corresponding to security level B3.
However, implementation of a commercial system is only a middle term objective. In the short term it has been decided to specify, design and implement a prototype version of the system. This prototype version will be satisfactory only if it demonstrates the feasibility of these main features of the system: universality, efficiency and power.
Therefore we will concentrate on the essential aspects leaving out aspects like user friendly interfaces, many data types handling, ...

Universality will be addressed thanks to the normalized audit file format (NADF) very similar to the BS2000 SAT file format, but still more flexible. This NADF format is universal in the sense that we believe that all existing (and future) audit trails can be translated into it in a sufficiently straightforward way. Audit trail analysis will be performed on normalized audit trails only. In addition, so-called *format adaptors* will be provided to translate native audit trails into normalized format. Feasibility of the approach will be demonstrated by providing format adaptors for the present existing audit trails on BS 2000 and SINIX.

Power will be provided by means of a kind of rule-based language allowing to express complex selection criteria dealing with arbitrary long sequences of records but also allowing to sequentially process the file *from left to right.* This last feature is of course mandatory to ensure efficiency as the amount of data to be processed is very huge. The basic principle is that the information about the past will be stored in a set of active evaluation rules that will be used to analyse the next audit record. Those rules will also possibly trigger off new rules for the analysis of the rest. Although powerful, this language will possibly be difficult to use because it involves a different style of programming. Therefore, it could be viewed as an intermediate language in which higher level predefined selection procedures could

be translated.
However, we will not address the definition and the translation of such higher level procedures due to the short duration of the project. Moreover, we will limit the variety of data types to be handled in selection conditions. Handling many data types will be very useful in the commercial system but should require much tedious work. Instead, we will use a single universal data type, namely string of bytes, that will allow to analyse any kind of data, at least in principle.

Efficiency will be provided on the one hand by the very principle of the rule-based language allowing to process each record only once and, on the other hand, by efficient implementation techniques. The C language will be used as well as an optimized evaluation of selection conditions and more generally of rule execution. We will extend here techniques already used in the SATUT evaluator. Therefore we plan to implement an evaluator as efficient as the existing SAT evaluator as far as recordwise selection is concerned. Moreover, we hope that it will also be efficient for much more complex selection criteria.

The rest of this report is organized as follows:
Chapter 2 gives a complete definition of the proposed rule-based language regardless of implementation issues.
Chapter 3 provides an overview of the prototype external functionalities.
Chapter 4 summarizes the main aspects of the rule-based language implementation (i.e. the rule evaluator).

Chapter 2

Definition of the rule-based language

2.1 Data types

The definition of the rule-based language must specify the available data types as well as the primitive operations defined for these data types. However, for the purpose of the prototyping phase and in order to keep this prototype as simple as possible, only two basic data types will be considered:

- string of bytes
- integer

In theory, every data can be represented as a string of bytes. Keeping data types as simple as possible is necessary to ensure compatibility with a large variety of audit trail formats. String of bytes is the main data type used in ASAX; the integer data type is introduced here to take advantage of arithmetic operations to specify complex selection operations on data records. These data types are available along with their common primitive operations.

2.2 Syntax

2.2.1 Lexical items

The second basic step in defining the ASAX language is the specification of its lexical items or tokens. We give here a BNF definition of what a lexical item is:

⟨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 ⟩

⟨X_character ⟩ ::= 0 | ... | 9 |A |B |C |D |E |F

⟨digit ⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨letter ⟩ ::= a | ... | z | A | ... | Z

⟨special symbol ⟩ ::= + | − | * | (|) | > | < | **!=** | ≤ | | ≥ | **:**
| **%=** | **;** | = | −> | **:=** | **,** | **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**

An ASAX program is a finite sequence of lexical items. This sequence is constrained to some syntactic rules here under defined:

2.2.2 Abstract syntax[1]

⟨rule declaration ⟩ ::= ⟨rule heading ⟩
⟨variable declaration part ⟩ ⟨action part ⟩

⟨rule heading ⟩ ::=
rule ⟨rule name ⟩ (⟨parameter list ⟩)

⟨rule name ⟩ ::= ⟨identifier ⟩

⟨parameter list ⟩ ::=
⟨empty ⟩
| ⟨parameter group ⟩ **;** **...** **;** ⟨parameter group ⟩
⟨parameter group ⟩ ::=
⟨parameter name ⟩ **, ...** **,** ⟨parameter name ⟩ **:** ⟨type ⟩

⟨parameter name ⟩ ::= ⟨identifier ⟩

⟨type ⟩ ::= **ascii_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 ⟩

[1] ⟨token⟩ *sep* ... *sep* ⟨token⟩ stands for a possibly empty list where *sep* is a separator.

$\langle$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 $\rangle$

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

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

$\langle$condition $\rangle$ ::= **true** | **false** | **present** $\langle$field name $\rangle$
  | $\langle$expression $\rangle$ $\langle$relational operator $\rangle$ $\langle$expression $\rangle$
  | $\langle$condition $\rangle$ $\langle$logical operator $\rangle$ $\langle$condition $\rangle$
  | **not** $\langle$condition $\rangle$

$\langle$relational operator $\rangle$ ::= $< \mid > \mid \neq \mid = \mid \leq \mid \geq$

$\langle$logical operator $\rangle$ ::= **and** |**or**

$\langle$repetitive action $\rangle$ ::=
  do $\langle$guarded action $\rangle$ **;** **...** **;** $\langle$guarded action $\rangle$ **od**

$\langle$compound action $\rangle$ ::=
  begin $\langle$action $\rangle$ **;** **...** **;** $\langle$action $\rangle$ **end**

$\langle$rule triggering $\rangle$ ::= **trigger off** $\langle$triggering mode $\rangle$ $\langle$rule call $\rangle$

$\langle$triggering mode $\rangle$ ::= **for_next**
  | **for_current**
  | **at_completion**

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

$\langle$predefined procedure name $\rangle$ ::= $\langle$identifier $\rangle$

$\langle$rule call $\rangle$ ::= $\langle$rule name $\rangle$ **(** $\langle$expression $\rangle$**,...** **,** $\langle$expression $\rangle$**)**

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

The semantics is defined wrt the abstract syntax not the concrete one. Concrete syntax is needed for parsing purposes only.
Full syntactic correctness of an expression, condition, construction depends on an identifier table gathering parameter names, variable names and field names. This table is simply called the *identifier table.*

2.2.3 Additional syntactic rules

1. Arithmetic operation apply to integer expression only;

2. Every identifier has to appear in the identifier table.

Abstract syntax will be used as a basis for the definition of the semantics of the rule-based language. It does not take into account parsing problems. In fact abstract syntax rules should be understood as defining tree structured objects, not just sequences of lexical items.
In a (future) commercial system, the rule-based language could be found too difficult to be directly used by the auditor. As we will see on examples, its power is due to a rather unusual programming style which can be found unatural and can be more difficult to control than usual programming. Therefore, the rule-based language could be used as an intermediate language: the auditor should use a simpler query language (à la SQL for example) but queries in this "higher" level language should be translated in a set of rule definitions of the rule-based language.
In such a system, there should be no need in theory for an "actual" textual version of the rule-based language. A query could be directly translated in internal form and such an internal form will be very close to the constructs defined by the abstract syntax if we ignore the optimization issues. (A tree is close to a data structure with pointers).
However, in the first stage of the project, we will need a textual version of the rule-based language in order to experiment on the language and test the rule evaluator. We think that implementing a parser for such a textual version is an easier task than defining a query language and a translator for it.
The textual version is also needed for writing examples on "paper". Finally, the rule-based language will be definitely more powerful than any query language implementable thanks to it. Therefore, the textual version will still be useful in the final system in order to write more sophisticated applications.

The *concrete* syntax of the language defines the actual way to write textual representation of the rules. Here, expressions are represented by sequences of lexical items, not trees. So the sequence

$$a + b * c$$

can be parsed in two different ways:

$$a + (b * c) \text{ and } (a + b) * c$$

corresponding to the abstract tree structured expressions in Figure 2.1.

The concrete syntax simply provides rules allowing to resolve such ambiguities. Those rules are in fact given by a more detailed BNF grammar that explicitely defines the parsing rules. They could be equivalently defined by priority rules for operators and associativity rules (for solving ambiguities between operators of the same priority). We will summarize those rules after giving the concrete BNF syntax.

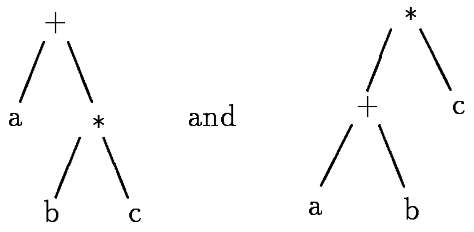

Figure 2.1: Parsing trees of a + (b * c) and (a + b) * c

2.2.4 Concrete syntax

⟨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	$<, >, \leq, \geq, =, \neq$
4	**not**, **present**
5	**and**
6	**or**

Note that the operators priority is implicit in the concrete syntax and needs not be stated here. This priority levels are chosen so that expressions are evaluated in a very natural way. For example an expression like:

$$a + b * c < c + d$$

is interpreted by the following parenthesised expression:

$$(a + (b * c)) < (c + d)$$

2.3 Semantics:

The aim of the present section is to give a meaning to each of the ASAX language constructs. We define here how expressions are evaluated, actions executed and rules triggered. Some basic concepts are first introduced to allow precise and concise explanations.

Current Record (CR) The execution of an ASAX program involves the analysis of an entire audit trail. At a given time of this execution, a certain set of active rules is processing one simple record. When all these rules have been executed with respect to this record, the next audit record is processed in turn. This processing is repeated until the end of the audit file is reached. The record being processed at a given time is called the *current audit record* or the *current record* for short. The current record is supposed to be a set of fields. A field is supposed to have two components[2]

- field name;

[2]Later, when addressing the audit record implementation (see 3.2.1) and in order to allow efficient access to audit data fields, each audit record field will have three components:

- length;
- identifier;
- value.

However, at the logical description level, it is sufficient to consider only two components.

- field value.

Current Environment(CE) Roughly speaking, the current environment is the set of information needed to describe precisely how the statements of the program are executed. Obviously, the current record is part of the current environment.
A current environment consists of the following:

- the current record;
- the local environment: a set of variables;
- DStrig: the set of *active* or *triggered* rules;
- DSnext: the set of rules to be triggered off for the next record;
- DScompl: the set of *completion* rules, i.e. the set of rules to be triggered off when the processing of the whole file is completed.

Rule schema
A *rule schema* is simply a rule declaration according to the syntax of paragraph 2.2.2.
Rule
A *rule* consists of a rule schema and a list of values to be substituted to every parameter name. This is very similar to procedure declaration and procedure call in Pascal.

2.4 Evaluation of expressions and conditions

2.4.1 Constants and literals

value of a literal
C and X_literals denotes string of bytes.
A byte is a sequence of eight bits. We identify bytes with the natural numbers they represent in binary. For example, the byte

$$11001110$$

is identified with the integer:

$$206$$

Let 'C_1, ..., C_n' be a C_literal where each C_i is either a printable ascii character with binary code B_i (a byte) or a sequence of two quotes (in this case, B_i will denote the ascii code of the quote character). By definition the above C_character denotes the string of bytes :

$$B_1, \ldots, B_n$$

Let

$$X'X_{1,1}X_{1,2}X_{2,1}X_{2,2}, \ldots X_{n,1}X_{n,2}'$$

be a 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)

2.4.2 Value of simple conditions involving the current record

Let *field_name* be a field name.

- The condition

  present *field_name*

  is evaluated to **true** if the current record contains a field name of name *field_name*; it is evaluated to **false** otherwise.

- The expression

  field_name

  is evaluated to the string s if s is the value of the field name *field_name* in the current record. It is evaluated to an arbitrary value if the current record does not contain a field name *field_name.* (This means that nothing can be assumed about this value but no error occurs. This convention will be useful for optimization purposes.)

2.4.3 Evaluation of arithmetic and relational expressions

Let

- E be the current environment;
- $expr_1$, $expr_2$ two arithmetic expressions;
- ω an arithmetic or relational operator.

The expression

$$expr_1 \ \omega \ expr_2$$

is evaluated with respect to E as follows:

1. $expr_1$ and $expr_2$ are evaluated wrt 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

$$-expr$$

is $(-v)$ where v is the value of the expression *expr* wrt E.
An integer constant is evaluated for itself.

2.4.4 Evaluation of compound conditions

Evaluation of boolean conditions is likely to be optimized as much as possible[3]. However, it is not desirable for the user (the auditor) to be aware of the exact optimizations because this would forbid all kinds of further 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:

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

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 ω 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 subconditions.
Note also that the value of *cond* will sometimes be defined while the value of a subcondition is not. For example, suppose that *foo* is a field name not present in the current record, then the condition

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

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,

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

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).

For **not** *cond*, the returned value is obviously the negation of the value returned for *cond*.

[3] Exactly as it is in the present SATUT evaluator

2.5 Execution of an action

2.5.1 skip statement

skip is the empty statement. It can be useful in clarifying the code. This is similar to the Pascal empty statement (";**skip**;"is equivalent to ";;").

2.5.2 Assignment

Let

- E be the current environment;
- *rexpr* a right expression;
- *lexpr* a left expression;

the execution of the action:

$$lexpr := rexpr$$

with respect to the current environment takes place in the following steps:

1. *rexpr* is evaluated with respect to the current environment E. Let v be its value (if this evaluation ever terminates);
2. *lexpr* is assigned the value v.

2.5.3 Conditional actions

Let

- E be the current environment
- $cond_1$, ..., $cond_n$ boolean expressions
- $action_1$, ..., $action_n$ a sequence of actions (n $\geq$1)

the execution of the statement:

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

is evaluated as follows:

1. $cond_1$ is executed with respect to E. Let v be its value.
2. if v = **true** the action $action_1$ is executed with respect to E and the execution is terminated; else

(a) if n > 1 the statement:

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

is executed with respect to the environment E.

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

2.5.4 Repetitive actions

Under the conditions of the previous paragraph, 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 E is as follows:

1. the conditions $cond_1, \ldots, cond_n$ are successively evaluated wrt E until one (if any) returns a true value. Let $cond_i$ $(1 \leq i \leq n)$ be such a condition. In this case, the action $action_i$ is executed and then 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 executed over again.

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

2.5.5 Compound actions

Let

- E be an environment
- $action_1$, ..., $action_n$ a sequence of actions (n > 0)

The execution of the statement:

$$\textbf{begin}\ action_1;\ ...;\ action_n\ \textbf{end}$$

with respect to the current environment E consists of successively executing each of the actions $action_1$, ..., $action_n$ with respect to E.

2.6 Rule triggering

Let E be the current environment defined by its components (see 2.3):

- the current record;
- the local environment;
- DStrig: the set of active rules;
- DSnext: the set of rules to be triggered for the next audit record;
- DScompl: the set of rules to be executed at completion time

Suppose that

rule *rule_name* (*parameter_list*);

local_variable_list;

rule_body

is the rule schema R, the execution of the action:

Trigger_off *triggering_mode* *rule_name* ($expr_1$, ..., $expr_n$) , ($n \geq 0$)

is as follows:

1. the expressions $expr_1$, ..., $expr_n$ are evaluated with respect to E. Let L $= \{v_1, ..., v_n\}$ be the list of their respective values;
2. (a) if *triggering_mode* = **for_current**
   then the effect of the execution is to add a new rule to the set of active rules:

   $$\text{DStrig} = \text{DStrig} \cup \{(\text{R, L})\}$$

   where R is the triggered rule identified by its name and its list of actual parameter values L.

   (b) if *triggering_mode* = **for_next**
   then the effect of the execution is to define a new current environment F identical to E except for DSnext which becomes:

   $$\text{DSnext} = \text{DSnext} \cup \{(\text{R, L})\}$$

   where R is the triggered rule identified by its name and its list of actual parameter values L.

   (c) if *triggering_mode* = **at_completion**
   then the effect of the execution is to define a new current environment F identical to E except for DScompl which becomes:

   $$\text{DScompl} = \text{DScompl} \cup \{(\text{R, L})\}$$

   where R denotes the triggered rule identified by its name and its list of actual parameter values L.

2.7 Predefined procedure call

Let E be the current environment. The execution of the predefined procedure call:

$$\textit{procedure\_name}\ (expr_1,\ \ldots,\ expr_n)\ ,\ (\mathrm{n} \geq 0)$$

is as follows:

1. the expressions $expr_1$, ..., $expr_n$ are evaluated with respect to E. Let v_1, ..., v_n be the values of $expr_1$, ..., $expr_n$ respectively;
2. the predefined procedure call :

   $$\textit{procedure\_name}\ (v_1,\ \ldots,\ v_n)$$

   is executed with respect to E.

2.8 Overview of the complete audit file processing

2.8.1 Initial environment

Suppose we have the n rules RI_1, ..., RI_n active at the initialization step (these rules are also called *initialization rules*) and RC_1, ..., RC_p, to be active at completion time.
Consider the following rule schema:

rule *init_rule* ;

begin

Trigger off for_next RI_1;

$$\vdots$$

Trigger off for_next RI_n;

Trigger off at_completion RC_1;

$$\vdots$$

trigger off at_completion RC_p

end;

The initial environment corresponding to these initial and completion rules is formed out of the following components:

- the first record of the audit trail;

- The local environment of the initial environment which is empty;
- DStrig = $\{init\_rule\}$;
- DSnext = $\{\}$;
- DScompl = $\{\}$.

Note

1. the initial environment should also include the description of the predefined procedures (contained in the library) as well as the description of some global variables such as the format adaptor table. However, for simplicity reasons, these additional initial environment components are made implicit.
2. the initial environment would have been defined by the following:
   - the first record of the audit trail;
   - an empty local environment;
   - DStrig = $\{RI_1, ..., RI_n\}$;
   - DScompl = $\{RC_1, ..., RC_p\}$;
   - DSnext = $\{\}$.

This initial environment is equivalent to the first one (in the sens that the analysis process will execute exactly the same way indifferently with respect to any of the two). However, the first initial environment have been retained for it has a canonical form (since it has a single initial rule). Furthermore, and although implementation issues are irrelevant at the specification level, the first environment would result in a more efficient implementation than the second one.

2.8.2 Complete audit file processing algorithm

inputs:

- the audit trail(s) to be processed in NADF format (the NADF format will be addressed in paragraph 3.2.1 of the next chapter);
- a set of rule definitions divided in three parts:

  DSinit: set of initialization rule definitions;

  DStrig: set of rule definitions, corresponding to rules to be triggered off dynamically;

  DScompl: set of completion rule definitions.

outputs: any kind of output generated during the processing.

At the beginning of the algorithm, DStrig is empty. The initial_rule is formed out of the elements of DSinit and DScompl as shown in the previous paragraph.

Algorithm

```
Scurr := init_rule;

Scompl := { };

Open the audit trail;

WHILE not end of audit trail

DO

    Begin
        read the next audit record;
        Process Scurr giving NSnext, NScompl;
        Scurr := NSnext;
        Scompl := NScompl ∪ Scompl
    End;

Close the audit trail;

Process Scompl.
```

Chapter 3

The prototype

3.1 Introduction

In the previous chapter, we described the specification of the rule-evaluator. This specification has been made independent of the underlying operating system and Audit Trail format. Assuming the system runs on UNIX and that audit trails have NADF format, the present chapter deals with the overall prototype executing environment. The arguments and arguments formats provided by the user are also discussed. Finally, the functioning of ASAX during a session is presented.

3.2 Security Audit Data Format (ADF)

Here under we explain what kind of format has been chosen as internal audit data format in ASAX and why. Then we present how ASAX will be able to analyse audit data of any format.

3.2.1 The Normalized Security Audit Data Format (NADF)

ASAX will process security audit records represented internally in a single format called *Normalized Security Audit Data Format* (NADF). This format is inspired by the BS2000 SAT audit record format, which has been proved efficient and flexible. However, it is more general.

An audit data record in the NADF format consists of a list of *audit data*, each of which being represented by:

- its identifier;
- its real length;
- its value.

The list is sorted according to the audit data identifier in order to increase the performance of the audit data processing.

3.2.2 The Format Adaptor

Internally, ASAX knows only the NADF format. However, as it is able to analyse different kinds of audit trails (likely to be represented in a different format) it has to transform them into the NADF format before processing them.
Before explaining what the format adaptor does, let us recall some significant features of audit trails.
In an attempt to reduce the size of audit trails, most of the audit mechanisms write some audit data in an internal form (also called *coded form*) using local customized naming conventions. Associated to this form, there is an external and/or global scope form (also called *readable form*) used to ask questions and selection criteria. (Example: the user-id is represented internally by an integer and externally by a string of characters).
The former approach is more appropriate when customizing aspects have to be solved.
These mapping rules are used to transform the question or selection criterion in an external form which can be compared directly with the audit record contents.
The ASAX format adaptor aims at converting any kind of audit trail in the NADF format, in a way as efficient as possible.
The original audit record to be converted is transformed in one or several audit record having NADF format[1]. An audit data is simply transformed into its identifier, its length and its value. Even if the audit data is represented in an internal/coded form in the original record, its value in the NADF audit record will remain coded. In fact, it is more efficient to keep the coded form of the audit data in the converted NADF format record than to transform it into the external representation since:

- the number of the audit records is likely to be much more higher than the number of selection criteria applied to each of them;
- the comparison between audit data and selection criteria is likely quicker when these two elements are represented in internal form;
- some coded values could be impossible to "decode" into external form;
- some coded values could be encrypted values not to be decoded for security reasons.

This also means that the task of the format adaptor is not restricted to the conversion of audit trails into NADF files. During the translation of a selection criterion into efficient evaluable internal form, ASAX will need the mapping rules between internal and external audit data representation.

[1]Some audit record may contain repetitive audit data i.e. its value is a list of n values. Because the rule based-language does not provide indexed variables or lists, this record must be splited into $n + 1$ audit records having NADF format. One of these records is the original record without this repetitive audit data; each of the remaining n records is formed out of a particular value (along with its length and identifier) of the repetitive audit data. This audit record spliting is performed by the format adaptor.

Thus the format adaptor also aims at storing the mapping rules between internal and external audit data representation.
In a first version, ASAX will contain for each supported kind of audit trail a specific program/routine able to realize the format adaptation. That means that the modification of the original audit trail format specification will require modifications of the format adaptor itself.
In further versions, ASAX will offer an interface allowing to describe various files specification and a parametrized format adaptor.
Nevertheless, in some special cases (for some very special new audit trail specifications) this more powerful tool could need some source code modifications.

3.3 The library of pre-programmed procedures

Not all kinds of procedures can be programmed in ASAX. The library contains a set of (pre-programmed) or predefined, ready to use procedures. Only the parameter description of these procedures is made visible to the user: the user do not have to bother about how these procedures have been implemented. Examples of these predefined procedures are sending a message pointing out an alarm, writing a selected record in the temporary file, printing a statistic table, etc.
Obviously, the procedure calls occurring in the set of rule schemata will be implemented by link editing the evaluator with the corresponding procedures of the library. The use of pre-programmed procedure calls can increase system functionalities. Furthermore, this library can be extended as needed to meet new system functionalities.

3.4 Session description

During an ASAX session, the user has to provide the following:

- the audit file to be analyzed in NADF format (given by its name according to naming conventions in use);
- the output file name (optional);
- the file (having an ASCII format) holding the table of correspondence between audit data identifiers and their code. This table is supplied by the format adaptor;
- a file containing a set of rule schemata;
- a special initial rule containing the actions to be executed in order to trigger off the initial rules (this rules are called DStrig in the previous chapter);
- the library of predefined procedures that can be referenced inside the rules.

The input file (i.e. the audit trail) is a file in NADF format. This file is supposed to have ASCII format. The file containing rule schemata is a text file.
These rules are first analysed and checked for syntactic and semantic errors. In absence of such errors, the system will store an internal representation of these rules. If these rules do reference some external pre-programmed procedures (and this is likely to be the case), these procedure codes are linked up with the rule evaluator. The result is a final executable program ready to run.
The initial rule is then executed and the audit trail opened. The analysis process can now start and the first record is analysed with respect to the current active rules triggered by the initial rule.
The output file is accordingly filled up with audit trail warning and advising messages. The audit trail administrator can then take the appropriate measures.

Note

In a future version, the analysis process can be suspended in order to allow the user to introduce new rules. This can be especially useful when the audit trail administrator suspects abnormal behaviours and wants to focus attention on these behaviours. More specific rules are then added to provide a closer analysis. However, this feature is rather complicated for the prototyping phase and is planed for a next version of the system.

Chapter 4

Examples of rules

4.1 Introduction

The following examples are presented in order to illustrate how the high level functionalities are implemented.

4.2 Example 1: failed login detection

Let us assume that the ASAX user has at his disposal the following alarm definition:
" If a user tries more than X times to log in the system from the same terminal without success within Y minutes, the message M_1 will be sent to the auditor "
This alarm function will be implemented using two rules:

`alarm_failed_login` and `Count_rule`

The first one will be activated with the appropriate values of parameters in order to achieve the required alarm.
The second rule will be activated by the first one.
The aim of the `alarm_failed_login` rule is to send a message to the auditor when the user unsuccessfully logs in the system the number of times *occ_par* from the same terminal within the period *within_par*. (between these unsuccessful logins, no successful login for this user from this terminal can appear).
It will be activated by the following parameters:

occ_par = X;

within_par = Y minutes.

The translation of this rule in the rule_based language will have the following form:

```
rule alarm_failed_login(occ_par, within-par: integer);

if evt = 'login'
  and res = 'f' → begin
                     Trigger off for_next
                        Count_rule (occ_par - 1,
                                    timestp + within_par,
                                    user-id,
                                    station);
                     Trigger off for_next
                        alarm_failed_login(occ_par,
                                           within_par)
                  end;
  true          → Trigger off for_next
                     alarm_failed_login(occ_par
                                        within_par)
fi
```

The aim of the `count_rule` rule is to send the message M1 (to the auditor) when the user `user-id-par` has unsuccessfully logged in the system the number of times `l-occ-par` from the terminal `terminal-par` and that the last unsuccessful login has occurred at a time < `limit-par` (between these unsuccessful logins, no successful login for this user and terminal can appear).
the translation of this rule in the rule-based language will have the following form:

```
rule count_rule (l-occ_par: integer;
                 limit-par, user-id-par, terminal-par: string);
if l-occ_par = 1
  and evt = 'login'
  and res = 'f'
  and userid-par = userid
  and station = terminal-par
  and timestp < limit-par    → send_message(M1);

  evt = 'login'
  and res = 'f'
  and userid-par = userid
  and station = terminal-par
  and l-occ_par > 1          → Trigger off for next count_rule
                               (l-occ_par - 1, limit-par,
                               user-id-par, terminal-par);

  timestp ≥ limit-par        → skip;
```

```
  res ='s'
  and evt = 'login'
  and station = terminal-par
  and userid-par = userid     → skip;

  true                        → Trigger off for next
                                    count_rule(l-occ_par,
                                               limit-par,
                                               user-id-par,
                                               terminal-par)
fi
```

In order to receive the message M1 each time a user fails to log three consecutive times from the same terminal within ten minutes, the auditor will activate the rule `alarm_failed_login` with 3 and 10 as values for the parameters (see [1] for more details).

4.3 Example 2: Detection of computer generated passwords

The purpose of this example is to detect two or more login attempts by the same user in such a short period of time that it is doubtful that they could have been generated by a human at a conventional computer keyboard. This rule is designed to detect the type of attack in which a penetrator uses a computer to repeatedly generate passwords / userid pairs. See [2].
Two rules are used to implement this alarm function:

`fast_login` and `timer`

The second rule is activated by the first one whenever a login attempt is encountered. The userid, terminal name and time stamp corresponding to this event are the arguments of the rule `timer`. It then acts like a timer. When a second login attempt for the same user and terminal occurs within a very short period (say 1 second), a message is sent to the auditor. Otherwise, the time interval is elapsed, the rule `timer` is desactivated.
Here is the translation of these rules in the rule_based language:

```
rule fast_login;
if evt = 'login'
  and res = 'f' → begin
                       Trigger off for next timer
                       (userid, station, timestp + 1);
                       Trigger off for next fast_login
                  end;
  true          → Trigger off for next fast_login
fi
```

```
rule timer(userid-par, terminal-par:  string; time:  integer);
if evt = 'login'
  and userid-par = userid
  and station = terminal-par
  and time > timestp          → send_message(M2);
  time ≤ timestp              → skip;
  true                        → Trigger off for_next timer
                              (userid-par,time)
fi
```

send_message(M) is a predefined procedure call that sends the message M to the auditor.

4.4 Example 3: Detection of logins through several terminals

This example involves the detection of two or more logins from physically different terminals. This function is useful to prevent users from monopolyzing too much system resources. See [3]. It is implemented by two simple rules:

first_terminal and second_terminal

Every time the first_terminal rule detects a login event, it triggers off the second_terminal rule that looks for another login event related to the same user but from a different terminal. Between these two events, no successful logout event for this user from the first terminal can appear.
The translation of these two rules in the rule_based language is the following:

```
rule first_terminal;
if evt = 'login → begin
                      Trigger off for next second_terminal
                      (userid, station);
                      Trigger off for next first_terminal
                  end;
  true          → Trigger off for next first_terminal
fi
```

```
rule second_terminal(userid-par, terminal-par:  string);
if evt = 'login'
  and userid-par = userid
  and station ≠ terminal-par → begin
                                 send_message(M3);
                                 Trigger off for next
```

```
                                        second_terminal
                                        (userid, station);
                                    end;
   evt = 'logout'
   and userid-par = userid
   and station = terminal-par  →  skip;
   true                        →  Trigger off for next
                                        second_terminal
                                        (userid, station)
fi
```

4.5 Example 4: Statistics

Let us suppose that we want to compute the mean value of remote login events per session. This quantity can provide an estimate of the amount of processing resources occupied by the local users. The problem is to compute the total number of login events and the total number of remote login events that occur during the session corresponding to these login events. The mean value is then the quotient of the two quantities.
We use two main rules to implement this statistic function:

`rlogin_statistics` and `rlogin_counter`

The aim of the first rule is to trigger off the `rlogin_counter` rule whenever a login event is encountered. It also increment the number of login events computed so far. Once triggered, the `count_rlogin` rule will compute the number of remote login events initiated during the session corresponding to this login event.
However, there is still another problem to be solved: once the analysis of the entire audit trail is completed, these two quantities will be lost. One solution to this is to write each of these numbers to a secure memory location and to retrieve it later. During the processing of the audit trail, when a new login or remote login event is found, this variable is incremented. Later, at completion time, a rule will be triggered off in order to retrieve these values and compute the mean value.
Suppose we have at our disposal three predefined procedures:

- `increment_login_number`;
- `increment_rlogin_number`;
- `compute_mean_value`.

The first two are responsible for incrementing the number of login and rlogin events. The third one computes the mean value at completion time. An additional rule `rlogin_mean_value` is used to call the predefined procedure `compute_mean_value`.
Therefore, the translation of these rules in the rule-based language is the

following:

```
rule rlogin_statistics;
if evt = 'login' → begin
                       increment_login_number;
                       Trigger off for next rlogin_counter
                       (userid, station);
                       Trigger off for next rlogin_statistics
                     end;
  true           → Trigger off for next rlogin_statistics
fi
```

```
rule rlogin_counter(userid-par, terminal-par:  string);
if   evt = 'rlogin'
     and userid = userid-par
     and station = terminal-par → begin
                                     increment_rlogin_number;
                                     Trigger off for next
                                        rlogin_counter
                                        (userid, terminal-par)
                                   end;
     true                       → Trigger off for next
                                     rlogin_counter
                                     (userid, terminal-par)
fi
```

```
rule rlogin_mean_value;
     compute_mean_value
```

Finally, the initialization rule *init_rule* (see 2.8.1) is the following:

```
rule init_rule;
     begin
        trigger off for_next rlogin_statistics;
        trigger off at_completion rlogin_mean_value
     end
```

Chapter 5

Overview of the rule_based language implementation

We only deal in this chapter with the implementation of the prototype described on chapter 3. Implementation of the format adaptor is not considered.
The standard audit file evaluator will consist mainly of a parser and of a rule evaluator linked with a set of pre-defined procedures. All programs and procedures will be written in C .

5.1 The parser

Will use standard and well-known parsing techniques that will not be recalled here. Efficient internal representations of rule schemata will be generated and stored in core memory for further use by the rule evaluator. Those internal representations are called the *internal code* in the sequel.
Internal code will consist of linked data structures implementing a postfix polish representation of the rules, which is known to be efficiently executable. Special attention will be devoted to the design of *condition internal code* because the rule evaluator will spend most of its time evaluating conditions. We will reuse the representation chosen for the SATUT evaluator which is likely to be optimal (see [1].)

5.2 The rule evaluator

The implemented rule evaluator will use the following objects:

- an audit file currently analyzed in NADF format;
- an internal representation of the *current record*, designed to allow a fast access to all its fields as many times as needed;
- the internal code of the rule schemata;

- for all triggered rules a memory area representing the local environment of the rules.

Let us explain some of those points in more details.

5.2.1 The current record

Before being analyzed, the current record will be copied in memory with its regular NADF format. Then it will be preprocessed before applying all active rules to it. The processing will be similar to what is presently done in the SATUT evaluator. In the rule internal code, access to a field of the current record will be supplied via an indirection table built at compile time. Each entry of the table will contain a field identifier and a pointer field inside the current record. The table is sorted with respect to the field identifiers. Therefore processing the current record will amount to scan in parallel the newly read record and the indirection table in order to update the pointer fields accordingly to the actual record format.

5.2.2 Local environment

At each execution step, there exist a set of triggered rules split into three sets: the *active* rules, the rules *triggered* for the *next* record, the rules *triggered at completion* time.
Conceptually, a triggered rule will consist of a rule schema and a list of actuals parameter values. This will be implemented by means of a memory area called the *descriptor* of the triggered rule. A descriptor will consist of:

1. a pointer towards internal code of the rule schema;
2. a data field for each parameter of the rule, containing the actual parameter value.

Actual parameter values will easily be accessed from the internal code by means of a basis + displacement method: the internal code will only contain the displacements parts; the basis is simply the descriptor address.
The three sets of triggered rules will be implemented by three linked lists providing easy way to add and delete descriptors, to switch from one set to another and so on.
Finally, exactly one rule will be processed at a time. Its processing will use an additional memory area called the *local area.* This area will be used for each rule execution and will therefore be allocated once for all.

5.2.3 The predefined procedures

Predefined procedures will be written in C and stored in a standard library. Moreover, the parser of the rule based language will be given a table describing each predefined procedure. Each entry of the table will consist of:

the procedure name, the type of each parameter, a pointer to an indirection table where the procedure address can be found. The rule evaluator will be linked with all the procedures of the table. At execution time of a rule, executing a predefined procedure call will amount to find the actual procedure address in the table and to call it. Note that the indirection table could in fact be implemented by means of a case statement.

Bibliography

[1] I. Mathieu. *Advanced Security Audit Analyser on Unix* 13.09.91. 40 pages

[2] F. Libion. *Towards "intelligent" Security Audit Trail Analysis Tools.*. MS Thesis. FUNDP Namur 1990-91. 130 pages.

[3] A. Baur, W. Weib. *Analysis of protocole data using AI techniques.* ZFE IS SOF 4. 07.11.89.

[4] SAT team. *Security Audit Trail step1.* 104.52.7 REV7. 22.11.91. 213 pages.