Detecting Breaches in Computer Security: A Pragmatic System with a Logic Programming Flavor*

Abdelaziz Mounji & Baudouin Le Charlier,

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

August 12, 1996

Abstract

Computer security is a topic of growing concern because, on the one hand, the power of computers continues to increase at exponential speed and all computers are virtually connected to each other and because, on the other hand, the lack of reliability of software systems may cause dramatic and unrecoverable damage to computer systems and hence to the newly emerging computerized society. Among the possible approaches to improve the current situation, expert systems have been advocated to be an important one. Typical tasks that such expert systems can achieve include evaluating the security level of a software configuration and detecting malicious or incorrect behaviors of users.

Logic programming provides a powerful formalism for knowledge representation and deductive reasoning and is therefore a good choice to build such expert systems. However general implementations of logic programming (e.g., Prolog) can be too complex and too inefficient to be used in a security context, where all users' actions should potentially be analyzed in real time.

In this paper, we describe a "hand-crafted" system especially designed for watching what is happening in a computer configuration on a real time basis. The system integrates ideas from logic programming, imperative programming and rule-based systems in a pragmatic way. Some aspects of the systems have been presented elsewhere; in this paper, we concentrate on a new component, which is based on the deductive data base approach.

1 Approaches to Computer Security

Securing computers and network computers against unauthorized access and misuse is a growing concern in the computer community as evidenced by the rising number of press reports about security incidents. There are several means of securing computers, which can be divided into two broad approaches. *Access control* provides mechanisms for preventing unauthorized external users

*Submitted to the Eight Benelux Workshop on Logic Programming. September, 1996. Louvain-La-Neuve, Belgium.

from accessing the system and for preventing insiders from accessing resources they are not allowed to use. *Monitoring Systems* attempt to overcome the weakness of the access control mechanisms by means of additional auditing procedures. Monitoring systems can be further divided into two categories. On the one hand, *Intrusion Detection Expert Systems* (IDES) attempt to detect anomaly and misuse behavior by analyzing security-related events occurring in a system, which are collected in a so-called *audit trail*. Ideally, IDES systems should be able to respond actively, in real-time, to break-ins while monitoring user activity. For instance, if an intruder succeeds in overcoming the password authentication mechanism, the IDES system should close the connection or alert the security administrator. On the other hand, *Configuration Analysis Systems* perform systematic analysis of the access control configuration in order to find possible security breaches that would make the system vulnerable to break-ins or misuse. While IDES systems dynamically monitor user activity to detect intrusions, access control configuration analysis systems examine security sensitive objects in the configuration and assess how the system can possibly be compromised. In other words, configuration analysis systems provide a snapshot of the security level by reporting a list of found flaws.

A great deal of research efforts have been devoted to establishing the foundations of intrusion detection systems [1, 3, 4] and a few operational systems have been developed [2, 7, 8, 10]. Many researchers have built effective access control configuration analysis systems [9, 13] and more recently, [14] analyzes network access control configuration. In the framework of intrusion detection, we developed ASAX (*Advanced Security audit trail Analysis on uniX*), a system for attack scenario detection that uses a rule-based language RUSSEL (*RUle-baSed Sequence Evaluation Language*), specifically designed for intrusion detection (see [5, 6, 7]).

In this paper, we propose an integrated approach by adding a configuration analysis component to our previous work on intrusion detection. This combination is desirable for two reasons. First, configuration analysis systems do not prevent a malicious user from creating and closing holes in the system security between successive snapshots. These holes may go unnoticed by the system administrator if s/he relies solely on this type of systems. Second, an IDES system may look for attack scenarios that are irrelevant because the configuration is in fact not vulnerable to these attacks. The integrated system is achieved by coupling ASAX with a configuration analysis component that uses a datalog-like language to represent the reasoning of an attacker and maintains a fact base that models the state of the configuration as it evolves in time.

The rest of the paper is organized as follows. The next section describes the overall architecture of the system. Section 3 presents the configuration analysis component and outlines its implementation. Section 4 contains the conclusion.

2 Combining Intrusion Detection and Configuration Analysis

In this section, we present the architecture of a system for security analysis consisting of an intrusion detection component coupled with a configuration analysis component (see Figure 1). Our system assumes the activity of an audit subsystem which is operating system dependent and produces a sequential file of audit records describing system events such as files creation or modification, session starting and closing, etc. An audit record is composed of several items characterizing the event. Audit records are converted from the native format into ASAX's canonical format NADF (*Normalized Audit Data Format*).

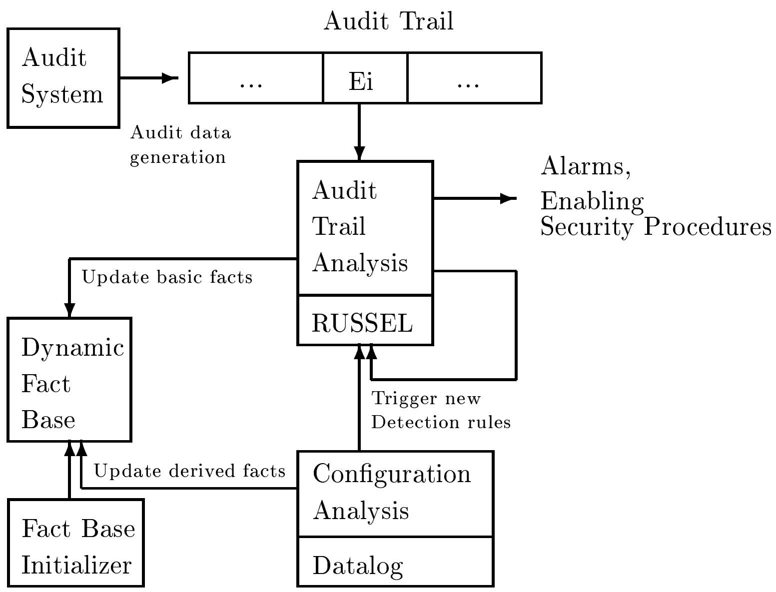

Figure 1: Combining Audit Trail Analysis and Configuration Analysis

2.1 Intrusion Detection with ASAX: the RUSSEL language

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. 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, output report messages, 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.

Example

The use of the RUSSEL language for audit trail analysis is best illustrated by a typical example: detecting access to security critical files and updating the fact base accordingly. In the example

```
rule file_access;
   begin
      if event in {CHMOD, DELETE, CREAT} and file_name in CRITICAL_FILES
         --> update_fact_base(event, file_name, own_id, own_gid, file_perm)
      fi;
      trigger off for_next file_access
   end;

init_action;
   begin
      trigger off for_next file_access;
      init_fact_base('datalog.log')
   end.
```

Figure 2: Detecting Access to Critical Files

(see Figure 2), the rule `file_access` examines each audit record of the audit trail in sequence. If the current record involves a critical file (e.g., `/etc/passwd` or a start-up file[1] such as `.login`), the external routine `update_fact_base` invokes the configuration component which updates the current fact base. Then, it retriggers itself for the next record. At the beginning of the analysis, the special rule **init_action** triggers the rule `file_access` for the first record, and also executes the routine `init_fact_base`. The latter compiles the datalog program `datalog.log`, and invokes the fact base initializer component which builds the initial fact base by browsing the actual configuration.

2.2 Access Control Configuration Analysis

To lay out the background of our discussion, let us first recall some basic concepts of the configuration under the Unix operating system. In Unix, users are identified by a number called their *user id* and denoted by *uid*; users are gathered into *groups*, which are similarly identified by a *group id*, denoted by *gid*. A file is owned by both a user and a group whose ids thus characterize the file. Each user grants three access rights (read-write-execute) to three sets of users (himself, his group, all users). In addition, users only exit through the processes they run. To every process are associated two identifiers: the user id and group id of the user who runs the process. When a process tries to access a file, the operating system checks the rights of the file against these identifiers.

Although the above principles constitute an effective way of preventing access to sensitive files and execution of privileged programs, an incorrect configuration of critical files may create security breaches. Such breaches may be especially difficult to prevent in large configurations of hundreds of users. As argued in [13], a possible way of addressing this complexity is to analyze the configuration from an attacker viewpoint. Clearly, a main goal of an attacker is to control execution of a highly privileged process. This can be done by adding some commands to the start-up files of a privileged user. This way, the attacker is able to execute the added commands under the identity (uid, gid) of the privileged user. (We say that the attacker *steals the identity* of the target user.) Furthermore, in order to access a given file, the attacker has to steal the identity of the owner of the file or of any

[1] A start-up file is automatically executed whenever a user logs in.

member of its group (if the owner grants access to the members). Alternatively, the attacker can attempt to obtain write access to the parent directory or to modify the authentication data base (/etc/passwd).

The above reasoning can be easily formalized using logic programming as depicted in Figure 3.

```
(1) become(U, V) :- start_up(V, F), write(U, F).
(2) become(U, V) :- write(U, /etc/passwd).
(3) write(U, F)  :- worldWrite(F).
(4) write(U, F)  :- groupWrite(F, G), groupMember(U, G).
(5) write(U, F)  :- parentDir(D, F), write(U, D).
```

Figure 3: Transitive Identity Stealing

The rules of Figure 3 express the following statements.

(1) If a user V grants access to one of his/her start-up files F to another user U, then the latter is able to steal the identity of (to "become") the user U.

(2) If a user has write access to the authentication data base /etc/passwd, then s/he is able to become any other user (including the super-user).

(3) If a file grants write access to all users, then any user has write access to that file.

(4) If a file grants write access to the members of the group, then any member of the group has write access to that file.

(5) If a user has write access to a directory, then s/he has write access to every file in this directory[2].

2.3 Overall Architecture of the System (Figure 1)

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

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

[2] In fact, he can remove and replace all files in the directory.

3 The Configuration Analysis Subsystem

3.1 The Deductive Language

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

3.2 Specification of the Inference Engine

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

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

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

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

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

During the first step, we remove all deductions to which some removed fact contribute. Simultaneously, we remove all derived facts that are no longer implied by any deduction. (Note that a fact can be implied by several different deductions.) During the second step, we use an incremental fixpoint algorithm (see [11, 12]) to compute all newly derived facts; additionally, we store the corresponding deductions.

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

Figure 4: Retracting a list of facts

3.3 Main algorithms

In this section, we present the main algorithms used in the datalog evaluation. Let `list_ded(f)` be the list of all deductions the fact f contributed to and let `Fact(d)` be the fact implied by the deduction `d`. The algorithm of Figure 4 computes the new fixpoint from a given Δ^{-}.

Moreover, we denote the number of deductions implying the fact `f` by `Nb_ded(f)`. We denote the set

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

by `rule_match`(f). The algorithm in Figure 5 computes the new fixpoint.

Given that the fact f matches the i^{th} atom of rule r, the subprogram `Gen_ded_fact(r, i, f, Δ)` in Figure 6, adds to Δ all facts that can be derived using all facts in the fact base. While executing this subprogram, the facts in $\Delta \setminus \{\mathtt{f}\}$ are not available for rule applications. The algorithm generates every possible derivation from all facts matching a given atom in r. Let us suppose that we already found a possibly empty list of facts $f_1,\ \ldots,\ f_{j-1}$ matching the atoms $a_1,\ \ldots,\ a_{j-1}$ of rule r and using the substitution α. The purpose of the algorithm in Figure 7 is to generate all deductions of the form $r\alpha\sigma$ such that

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

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

In the algorithm of Figure 7, we distinguish three cases:

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

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

Figure 5: Generation of all deductions from a given rule and a fact

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

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

3.4 Low Level Implementation

In order to make the incremental algorithms efficient, we need to store every fact and every deduction into separate data structures. To compute `list_ded(f)` efficiently, each fact provides a direct access to all deductions to which it contributes while every deduction provides an access to the fact resulting from the deduction (`Fact(d)`). In addition, to compute `rule_match(f)`, each predicate provides access to all rules where it appears in the body. Moreover, every fact contains a counter of the number of deductions from which the fact results. Unicity of representation of every fact and deduction is ensured by means of hashing techniques.

```
Gen_ded_fact(r, i, f, Δ)
begin
    Let r ≡ h -: a_1, ..., a_n;
    Let σ = mgu(a_i, f);
    Gen_case(r, σ, i, 0, Δ)
end.
```

Figure 6: Generation of all deductions from a given rule and a fact (*continued*)

```
Gen_case(r, α, i, j, Δ)
begin
Let r ≡ h :- a_1, ..., a_n;
if (j = n+1) then
    begin
        if (rα ∉ SD) then
            begin
                SD := SD ∪ {rα};
                if (hα ∈ BF ∪ DF) then begin DF := DF ∪ {hα}; Δ := Δ ∪ {hα} end
                else increment_ref(hα)
            end
    end
else if (j = i) then Gen_case(r, α, i, j+1, Δ)
else begin
        list_facts := find_all_facts(a_jα);
        while (list_facts ≠ ∅) do
            begin
                Remove(list_facts, f');
                σ := mgu(f', a_jα);
                if (σ ≠ fail)
                then Gen_case(r, ασ, i, j+1, Δ)
            end
    end
end.
```

Figure 7: Generation of all deductions from a given rule and a fact (*end*)

4 Conclusion and Future Works

We have presented the deductive component of a programming system that aims at analyzing a computer configuration as well as its users' behaviors efficiently (in real time). We use a specific —hand-made— implementation of the logic language since we believe that such specialized implementation is needed to attain our efficiency goals. Future works will be devoted to assess the practicality of our system for monitoring various aspects of network management and computer security in a real life context. In particular, we will develop methods to program and tune such system systematically.

References

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] D. Farmer and E. H. Spafford *COPS Security Checker System*. Purdue University, West Lafayette, Indiana 47907-1398, Sep 1991.

[10] P. Helman, G.E. Liepins, and B. Rochards, *Foundations of Intrusion Detection*. Proceedings of the Fifth Computer Security Foundations Workshop, June, 1992

[11] Michael J. Maher, Raghu Ramakrishnam *Deja vu in Fixpoints of Logic Programs* Computer Sciences Technical Report Num. 893. Computer Sciences Department, University of Wisconsin-Madison. November 1989.

[12] Raghu Ramakrishnam, Divesh Srivastava, S. Sudarshan *Efficient Bottom-up Evaluation of Logic Programs*. Computer Sciences Department, University of Wisconsin-Madison. November 1989.

[13] Baldwin Robert W. *Kuang: Rule-based Security Checking*. MIT, Lab for Computer Science Programming Systems Research Group. May 1994.

[14] Wietse Venema *Project Satan: UNIX/internet security*. Proceedings of the COMPSEC'95 conference, Elsevier, London, 1995.