

# Handling Generic Intrusion Signatures is not Trivial

*Jean-Philippe Pouzol and Mireille Ducassé*

IRISA/INSA de Rennes, Campus Universitaire de Beaulieu  
F - 35042 Rennes Cedex, France, email: {pouzol,ducasse}@irisa.fr

29th April 2000

## 1 Introduction

This article presents work in progress in the context of misuse scenario detection, where the scenarios are combinations of several actions. An example of a masquerading scenario is: “users manage to usurp the identity of someone, say  $x$ , then they copy the file `/bin/sh` and change its access rights such that later executions of the copy will give the privileges of  $x$ <sup>1</sup>.”

An intrusion detection system (IDS) based on a misuse detection strategy searches audit trails for *signatures* which are the traces of intrusion scenarios. Whereas the scenarios represent the point of view of the intruders, the signatures are what can be seen on the attacked system. We will say that signatures corresponding to scenarios involving several actions are *multi-event signatures*.

Signatures can be generic ; in our previous example, in whatever file the users copy the sensitive file does not make any difference, the name of the usurped user does not make any difference, either, at least for the detection. The successful break-in is very difficult to detect, especially if the password has been simply stolen. Therefore a trace of this scenario execution consists mainly of the copy of the file and of the change of access rights of the copy. A signature could be informally specified as: “the file `/bin/sh` has been copied on a target file *target* by a user  $u$ ; then the same file *target* has had its access rights changed by the same user  $u$ .”

Generic signatures are mandatory in a powerful IDS. Indeed, for the above example, in the one hand, it is impossible to foresee all the possible names for the target file. On the other hand, ignoring this target name makes the signature ineffective: signaling all the changes of access rights to any file will generate too many false positives.

---

<sup>1</sup>This should no longer be an effective attack, however detecting such a behavior would help to catch a potentially dangerous person.

<b>Filter</b>	$F ::= [t_1, t_2, \dots, t_n]$ $t_i ::= attr\_name = term$ $term ::= const\_val$   $var\_name$	<b>Signature</b>	$S ::= F$   $S \text{ or } S$   $S \text{ then } S$   $S \text{ and } S$   <b>var</b> $var\_list$ <b>in</b> $S$ <b>end</b>
---------------	---	------------------	--

Figure 1: Syntax of a simplified signature description language

From an operational point of view, generic signatures require some care. When the IDS is searching for occurrences of the above signature, whenever somebody is copying file `/bin/sh`, the variable *target* will be instantiated in the current occurrence of the signature. However, other instances of the same attack could be going on at almost the same time, and the file `/bin/sh` could be copied to other targets. In order not to be fooled by the beginning of the first attack, the IDS has to make sure that as many instances of the signature as needed can be generated. This is particularly a problem when the signature contains possibly intertwined sequences as shown in Section 3.

Whereas some systems already address this issue (such as [2]) others (such as [3]) rely on their users to handle it. It is important that the problem is analyzed independently of implementation details such that solutions can be automated to any system.

In the following, Section 2 first specifies informally a simplified language to describe signatures. This language does not pretend to cover all the needs for signature specification. It is, however, sufficient to illustrate the main topic of this article. Section 3 then illustrates in more detail the problems of handling generic multi-event signatures. Section 4 gives a sketch of solution which enables an IDS to be built on top of a concurrent search engine.

## 2 A simplified language to describe signatures

An IDS takes as input an *audit trail* which is a sequence of *events*. An event is basically a data structure with fields identified by *attribute names*.

An IDS based on a misuse detection strategy searches the sequence of events for instances of specified *signatures*. A signature is an abstract specification of the manifestation of an intrusion scenarios. A signature is basically a combinations of *filters*. A filter specifies conditions under which an event of the audit trail will be of interest.

In our simplified language, as specified in Figure 1, a filter is a tuple of items of the form  $attribute\_name = term$ , where a term can be a constant or a variable name. For example,  $[cmd = cp, arg1 = /bin/sh]$  is a filter which specifies that events whose command is `cp` and whose first argument is `/bin/sh` are of interest, and this independently of their other fields.

A single-event signature is described by a single filter. A multi-event signature

combines filters with three operators, **or**, **then** and **and**. Variables allow generic signatures to be specified. An informal description of the operators follows.

- *Single filter* ( $S = F$ ):  $S$  is detected if an event matches the filter  $F$ .
- *Choice* ( $S = S_1$  **or**  $S_2$ ):  $S$  is detected if either  $S_1$  or  $S_2$  are detected.
- *Sequence* ( $S = S_1$  **then**  $S_2$ ):  $S$  is detected if  $S_1$  is detected, and if  $S_2$  can be detected in the remaining part of the audit trail.
- *Intertwining* ( $S = S_1$  **and**  $S_2$ ):  $S$  is detected if  $S_1$  and  $S_2$  can be independently detected in the trail. In other words, we only impose a partial order on events which compose the instance of the signature. Note that, as opposed to the sequence, **and** is symmetrical.
- *Genericity* ( $S = \mathbf{var\ var\_list\ in\ } S \mathbf{\ end}$ ): the semantics of this statement depends of the construction it is applied to:
  - **var**  $V$  **in** ( $S_1$  **or**  $S_2$ ) **end**: the two branches of an **or** do not share variables. This statement can be rewritten as  $(\mathbf{var\ } V \mathbf{\ in\ } S_1 \mathbf{\ end})$  **or**  $(\mathbf{var\ } V \mathbf{\ in\ } S_2 \mathbf{\ end})$ .
  - **var**  $V$  **in** ( $S_1$  **then**  $S_2$ ) **end**: If  $S_1$  does not depend on  $V$ , the signature can be rewritten  $S_1$  **then**  $(\mathbf{var\ } V \mathbf{\ in\ } S_2 \mathbf{\ end})$ . Otherwise, the value bound to  $V$  while matching  $S_1$  is used to instantiate  $S_2$ .
  - **var**  $V$  **in** ( $S_1$  **and**  $S_2$ ) **end**: If  $S_1$  (resp.  $S_2$ ) does not depend on  $V$ , the statement can be rewritten  $S_1$  **and**  $(\mathbf{var\ } V \mathbf{\ in\ } S_2 \mathbf{\ end})$  (resp.  $S_2$  **and**  $(\mathbf{var\ } V \mathbf{\ in\ } S_1 \mathbf{\ end})$ ). Otherwise, the signature is matched if both  $S_1$  and  $S_2$  can be matched with the same value bound to  $V$ .

The signature informally specified in the introduction as “The file `/bin/sh` has been copied on a target file `target` by user `u`; then the same file `target` has had its access mode changed by the same user `u`.” can now be formally specified by

```

S = var TARGET in
    ([cmd = cp, arg1 = /bin/sh, arg2 = TARGET, usr = USER]
    then
    [cmd = chmod, arg1 = 4777, arg2 = TARGET], usr = USER)
end

```

### 3 Variables and concurrent searches of signatures

Figure 2 illustrates some delicate situations when searching for signatures containing both variables and intertwining. It shows a signature  $S$  with a variable  $V$ , and 3 audit trails. We use the following notation:  $F_i[V]$  is a filter constrained by a variable  $V$ ;  $E_i(\alpha)$  is an event matching  $F_i$  and instantiating  $V$  with the value  $\alpha$ ; “...” stands for an arbitrary number of events which do not match any  $F_i$ .

$$S = \left( \text{var } V \text{ in } (F_1 \text{ then } F_2[V]) \text{ and } (F_3[V] \text{ then } F_4) \text{ end} \right) \text{ then } F_5$$

**Trail A:** An audit trail with two instances of  $S$  where  $V$  is respectively valued to  $\alpha$  and  $\beta$   
 $\dots E_1 \dots E_2(\alpha) \dots E_3(\alpha) \dots E_4 \dots E_5 \dots E_3(\beta) \dots E_4 \dots E_2(\beta) \dots E_5 \dots$

**Trail B:** An audit trail with one instance of  $S$  where  $V$  is valued to  $\alpha$   
 $\dots E_1 \dots E_3(\alpha) \dots E_2(\alpha) \dots E_4 \dots E_5 \dots E_2(\beta) \dots E_3(\beta) \dots E_5 \dots$

**Trail C:** An audit trail with two instances of  $S$ , where  $V$  is respectively valued to  $\alpha$  and  $\beta$   
 $\dots E_1 \dots E_2(\alpha) \dots E_2(\beta) \dots E_3(\alpha) \dots E_3(\beta) \dots E_3(\gamma) \dots E_4 \dots E_5 \dots E_2(\gamma) \dots$

Figure 2: A signature  $S$  with a variable  $V$ , and 3 audit trails

Audit trail **A** contains two instances of  $S$ :

$$(E_1; E_2(\alpha); E_3(\alpha); E_4; E_5) \quad \text{and} \quad (E_1; E_3(\beta); E_4; E_2(\beta); E_5)$$

During the search process, when  $V$  is instantiated by  $E_2(\alpha)$ , the IDS must concurrently search for the remaining part of  $S$  with  $V = \alpha$ , and for another instance of  $S$ . Since  $F_1$  does not depend on  $V$ , it can be shared by the instance with  $V = \beta$ . The IDS should not search again for an event matching  $F_1$ .

Audit trail **B** contains a single instance of  $S$  with  $V = \alpha$ :

$$(E_1; E_3(\alpha); E_2(\alpha); E_4; E_5)$$

This trail does not contain an instance where  $V$  takes the value  $\beta$  because there is no event  $E_4$  after  $E_3(\beta)$ . The filter  $F_4$  does not depend on the variable  $V$ , but an event satisfying  $F_4$  is not necessarily shared by all instances of  $S$ . Indeed, filter  $F_4$  is part of a sequence, it has to be found after the previous part of the sequence, which here depends of  $V$ .

Audit trail **C** contains two instances of  $S$ :

$$(E_1; E_2(\alpha); E_3(\alpha); E_4; E_5) \quad \text{and} \quad (E_1; E_2(\beta); E_3(\beta); E_4; E_5)$$

In the trail, event  $E_5$  belongs to both instances of  $S$  because it occurs after all the last events of the intertwining branches of the two instances for  $\alpha$  and  $\beta$  have been reached. On the other hand,  $E_5$  occurs before  $E_3(\gamma)$ , and cannot be used to instantiate  $S$  with  $V = \gamma$ . This trail does not therefore contain an instance where  $V$  takes the value  $\gamma$ .

The first two trail examples show that the location of the filters which may instantiate  $V$  is significant to decide if an event can be shared between two instances of  $S$ . The intertwining of two sequences leads to difficulties because when one branch instantiates  $V$ , it is hard to know the progress in the other branch.

The third trail example illustrates that **and** operators define synchronisation barriers in the search process. The two parts of the signatures are matched

independently, and the search for the remaining part of the signature is started when both branches have reached the barrier ( $F_5$  in the signature example). Since the continuation after an **and**-construction could be another composed construction, it is not straightforward to express where to join the two branches of the **and** and when the search of the remaining part of the signature must be started.

## 4 A normal form for signatures

We saw in the previous section that it is not trivial to handle concurrent searches in presence of variables. This is especially true for signatures with intertwining patterns. We also specified in Section 2 that the choice pattern is simple to handle because the two branches do not share variables.

Therefore, we propose to rewrite the signatures in a normal form which is simpler to handle. Intuitively, this normal form is a disjunction of sequences of filters. When a signature is in the normal form, all **and**-constructions have been suppressed. For example:

$$(A \text{ then } B) \text{ and } C \implies \begin{array}{l} C \text{ then } (A \text{ then } B) \\ \text{or } A \text{ then } (C \text{ then } B) \\ \text{or } A \text{ then } (B \text{ then } C) \end{array}$$

Signature rewriting can be performed using inductive rules on the syntactical constructions. We give an example of such a rule in the case of a sequence pattern. Given a signature ( $S_1 \text{ then } S_2$ ), where  $S_1$  is in a normal form ( $S_{11} \text{ or } \dots \text{ or } S_{1n}$ ), the following rule can be applied:

$$(S_{11} \text{ or } \dots \text{ or } S_{1n}) \text{ then } S_2 \implies (S_{11} \text{ then } S_2) \text{ or } \dots \text{ or } (S_{1n} \text{ then } S_2)$$

The main advantage of this normal form is that we can add variables to signatures and easily describe what concurrent searches must be performed to find instances of signatures in the audit trail. Informally, all the searches can now be viewed as independant threads with their own variables. And when a variable needs to be instantiated, the thread is forked: the child thread will search for the remaining part of the sequence with the variable instantiated in the child context; the current thread will restart to search for an other possible instantiation of the variable in the trail.

Unfolding all possible intertwinings into sequences leads to a combinatorial explosion. The limited size and complexity of known scenarios, however, make us believe that this explosion can still be managed.

## 5 Discussion and future work

Some systems already handle variables in intrusion signatures. Among others, Kumar proposed in his PhD thesis to use Colored Petri Automaton (CPA) with

guarded transitions [2]. In this model, the colour of a token represents a possible valuation of the variables which appeared in the signature. The concurrent searches are performed by duplicating a token when a variable is instantiated. CPA is also a good model to describe intertwining patterns in signatures. Indeed, it allows several places to be linked to the same transition ; this naturally builds synchronisation barriers.

The representation of signatures proposed by Kumar is concise and expressive, but relies on a particular abstract machine. CPA with guards offer many primitive constructions, and to use this representation in another context requires to develop a CPA simulator. When signatures are rewritten in the normal form we proposed in Section 4, the only primitive needed to perform concurrent search is a *fork-like* instruction, which is available in all concurrent systems.

This preliminary work on a normal form which simplifies the management of variables in signatures will be pursued in three directions. We will first formally describe the normal form and the transformation rules. In so doing, we will verify that signatures in the normal form are equivalent to the original. We will, next, study optimizations to be applied to perform the concurrent searches. Each thread is conceptually seen as an independant process and this vision is effective to prove the correctness of the normal form. However, from an implementation point of view, it is possible to group some searches and dispatch information between threads. Finally, we want to extend the simplified signature description language with other primitives such as loops, or negation.

In parallel, we plan to concretely verify our results by developing a first prototype of a compiler which will produce code for the ASAX system. ASAX is a rule based misuse IDS proposed by Mounji [3]. Its rule based language provides the fork-like primitive we need. Moreover, a benchmarking study on ASAX is currently performed at IRISA/INSA, and ASAX seems to be a valid candidate to experiment our work in a real life scale [1].

**Acknowledgment** We thank Erwan Jahier for fruitful discussions.

## References

- [1] V. Abily and M. Ducassé. Benchmarking a distributed intrusion detection system based on ASAX: Preliminary results. Submitted to RAID 2000.
- [2] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, 1995.
- [3] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix - Namur (Belgique), 1997.