Enforcing Obligation with Security Monitors

Carlos Ribeiro, André Zúquete, and Paulo Ferreira

IST/INESC R. Alves Redol N°9 1000 Lisboa, Portugal
Carlos.Ribeiro@inesc.pt

Abstract With the ubiquitous deployment of large scale networks, more and more complex human interactions are supported by computer applications. This poses new challenges on the expressiveness of security policy design systems, often requiring the use of new security paradigms. In this paper we identify a restricted type of obligation which is useful to express new security policies. This type of obligation includes the following general situations: i) when two or more actions oblige each other, i.e. if one action is executed the others must also be executed and reciprocally, and ii) when an action obliges another and the obligatory action is causally dependent on the first action.

1 Introduction

The growing number of Internet users and services raises constantly new challenges for defining and ensuring adequate security policies. Most policies implement solely access control barriers, based on the concepts of permission or prohibition, but the current expansion of electronic business will stress, in a near future, the needs for more sophisticated security policies. In particular, we believe that the concept of obligation will have an increasing importance for the expressiveness of such policies. The need for ensuring obligation has already been recognized by several authors [1,2,3] and is illustrated by the following examples.

Consider that Alice browses through a site where she acquires several goods, when she leaves the site she is *obliged* to pay for the goods she acquired, otherwise the goods are not bought. Usually this policy must be enforced within the site's code, because the security service cannot enforce this kind of policy. Another illustrating example is when Alice registers herself, via a web server, as a student of *Online University*. Once she has done that, she is *obliged* to register herself as a student of, at least, a discipline chosen from a set of available disciplines. On the other hand, Alice could first register herself in a discipline; in this case she is then obliged to register as a student of the *Online University*.

These examples show that there is a clear need for expressing an application-specific obligation in a flexible way; and enforcing obligations with a security monitor has obvious advantages: it is language and application independent, and can be found in a large number of environments (virtual machines, operating systems, etc.).

2 Enforceable Obligations

To act upon security policies, a security service must know when someone attempts to violate those policies and what to do when that happens. On most security services, the attempts to violate rules based on permission and prohibition concepts are detected when an event requesting an action occurs and, in that case, the action requested is denied. The difficulty with rules based on obligation is that the time at which a violation attempt occurs and the action to perform when that happens are not so easy to instantiate on a particular instant and action, respectively. First, because a generic obligation does not need to have a deadline and second because there is not a generic action (equal for every situation) to perform in case of violation attempt.

Fortunately, obligation rules are seldom generic. Often what a security manager wants to express is "Conditional Obligations", in which obligations are triggered by pre-condition events: "*U1* `must do` *O* `if` *U2* `has done` *T*". While with the generic type of obligation a system is in an unsafe[1] state until the obligation has been fulfilled, with the conditional obligation a system has two safe states, one before the triggering event (T) and one after the obligation (O) is fulfilled. Thus, on the impossibility of fulfilling the obligation the system may always return to the safe state before the activating event, i.e. undo T. However even conditional obligations cannot be enforced solely by a standard security monitor. Using simple logic[2] it is possible to rewrite the conditional obligation expression into an expression with a dependency on a future event: "*U2* `cannot do` *T* `if` *U1* `will no do` *O*".

Schneider [4] states that with a monitor it is not possible to enforce a security policy in which the acceptability of an event depends on possible future events. Informally, his argument is quite simple: given the executions (sequence of events) τ and τ', in which τ is the prefix of some execution τ', it is not possible to allow τ on the basis that one of its extensions τ' is allowed by the security policy, because the system could stop before τ', and the system would have failed to enforce the policy.

The key issue that differentiates our work from Schneider's is the underlying model of execution. While to Schneider a system evolves through units of execution controlled by the security manager, which are independent from each other, to us those units may be organized in atomic sequences, thus depending on each other. By atomic we mean, in the sense of transactions' ACID properties, either all happens or none happens. Inside these atomic sequences of execution it is possible to define security policies with dependencies on future actions, because it is not possible for a system to stop execution leaving the sequence incomplete.

There are several ways to implement transactions [5], namely by keeping an undo-log with the information needed to reset the system to the initial state in case of failure, or by defining compensating actions for those actions that

[1] Unsafe in the sense that the security policy has not been completely enforced until then.

[2] $O \Leftarrow T \equiv \neg T \Leftarrow \neg O$

cannot be undone but can be compensated. However, there are some actions that cannot be undone or compensated, e.g. sending a document to a printer. These actions are called *real actions* on transaction management systems [5] and are already known to require special treatment by those systems in order to achieve atomicity. Implementing security obligations within transactions increases the number of *real actions*, because these must include actions that change human knowledge state (e.g. showing some text on the screen), which are not dealt by most transactional management systems.

3 Implementing Enforceable Obligations

We have implemented the obligation concept within our access control framework. This framework is composed by a security policy language (SPL) and its compiler[6]. SPL is a security language designed to express policies that aim at deciding about the acceptability of events.

An SPL policy is a structure composed of sets and rules, whose purpose is to express simple concepts like "separation of duty", "information flow", or "general access control". Sets contain the entities used by the policies to decide on events acceptability. A rule is a function of events, and may assume three values: "allow", "deny" and "notapply". Its purpose is to decide on the acceptability of the current event. A rule can be simple or composed. A simple rule is a tuple of two logical expressions. The first logical expression decides on the applicability of the rule, and the second decides on the acceptability of the event. Each policy has one special rule called the "query rule", which is identified by a question mark before the name, whose purpose is to define the policy behavior.

A simple policy stating that documents internal to the organization defining the policy cannot be sent to someone outside the organization, can easily be expressed in SPL:

```
policy Private( user set OrganizationUsers ) {
  object set InternalDocs:                                  // Policy data
?Private:                                                   // Rule name.
  ce.action = "SendEmail" & ce.target IN InternalDocs // Applicability exp.
  :: ce.parameter[1] IN OrganizationUsers                   // Acceptability exp.}
```

The rule uses the special variable "ce" to access the current event properties. The applicability expression of the rule states that the policy is defined only for events whose targets are documents internal to the organization and whose action is to send an Email. The acceptability expression states that for those events that satisfy the applicability expression the only events allowed are the ones that send the Email to a user inside the organization.

Given the future-dependent nature of obligation-based policies, they are expressed in SPL by quantifying a variable over the special abstract set `FutureEvents`, which encompasses all the events they are to be performed after the current event. Figure 1 shows an example of an information flow policy which uses obligation to force applications to register the information flow originated by them

```
policy InfoFlow () {

interface ReadFlowActions,
interface WriteFlowActions;
collection ProtObjects;

?InfoFlow:
  EXISTS fe IN FutureEvents {
   FORALL pe IN PastEvents {
    FORALL g IN pe.target.groups {
      ce.action IN WriteFlowActions &
      ce.task = pe.task &
      pe.target IN ProtObjects &
      pe.action IN ReadFlowActions &
      :: ce.target IN g } } }; }
```

(a)

```
policy HistoryInfoFlow () {
interface ReadFlowActions, WriteFlowActions;
collection ProtObjects;
?InfoFlow:
 FORALL te IN PastEvents {          // (1)
  EXISTS fe IN PastEvents {
   FORALL pe IN PastEvents {
    FORALL g IN pe.target.groups {
     ce.action.name = "commit" &   // (2)
     ce.trans_id = te.trans_id &   // (3)
     te.time < fe.time &           // (4)
     te.time > pe.time &           // (5)
     te.action IN WriteFlowActions &
     pe.target IN ProtObjects &
     pe.action IN ReadFlowActions &
     pe.task = te.task
     :: te.target IN g } } } }; }
```

(b)

Figure 1. (a) An information flow policy. (b) The transformation into an history-based policy

into SPL rules. This policy is not a strict information flow policy in the sense that it cannot handle implicit flows, as defined in Denning [7]. However, in some situations [8] the information leak resulting from implicit flows does not pose a serious security risk, either because the information on variables determining the sequence of execution is public or because it is not possible to infer the sequence of executions from the results of that sequence. For these situations it is possible to define information flow policies enforceable by event monitors, because the regulation of explicit information flow, from storage to storage, can be performed with just the knowledge on past events properties.

As explained in Sect. 2, the problem of enforcing obligation-based security policies is reduced to allowing or not the event that instructs the transaction monitor to *commit* a transaction, whether or not every obligation was fulfilled at the time of that event. A security policy that allows or denies an event (the commit event) depending on whether or not some events were executed (the obligations) is a history-based policy. In [6], we have shown that history-based policies can be efficiently implemented using special tuned logs for each policy, thus obligation-based can also be implemented efficiently in the same way.

The transformation from the obligation-based policy to the history-based policy can be achieved in two steps. The first step, called "aging", consists of replacing references to events by older references: (i) References to the current event are replaced by references to a past event called "trigger-event" (line (1) of Fig. 1b) ; (ii) References to past events are replaced by references to other

past events with an additional constraint specifying their occurrence before the trigger-event (line (5) of Fig. 1b); (iii) References to future events are replaced by references to past events with the additional constraint of occurring after the trigger-event (line (4) of Fig. 1b). The second step consists of inserting in this policy an explicit reference to the event that requests the transaction commit (lines (2) and (3) of Fig. 1b).

Due to space limitations we defer the details on performance of history-based policies to [6]. Nevertheless, the important observation is that, on all tests performed the delay on the commit-event caused by the information flow policy was in the worst case less than $1ms$, which is negligible compared to the actual commit time[3].

4 Conclusion

We have identified a restricted type of obligation which is simultaneously useful to express the security policies of large organizations and can be enforceable by security monitors. This type of obligation includes the following generic situations: i) when the two actions involved in a conditional obligation oblige each other, and ii) when the obligatory action is causally dependent on its trigger action. Our approach consists on using the transaction concept to delay the actual security monitoring until the commit time; thus, avoiding the problem of future dependency inherent to any obligation policy. We have developed a security language and a compiler encompassing the obligation paradigm, and the performance results show that it can be efficiently implemented.

References

1. Jonscher, D.: Extending access control with duties - realized by active mechanisms. Database Security, VI: Status and Prospects. (1992) 91–112
2. Cuppens, F., Saurel, C.: Specifying a security policy: A case study. In: IEEE CS Computer Security Foundations Workshop (CSFW96). (1996) 123–135
3. Marriott, D., Sloman, M.: Implementation of a management agent for interpreting obligation policy. In: IEEE/IFIP 7th Int. W. on Distributed Systems Operations and Management, Italy (1996)
4. Schneider, F.B.: Enforceable security policies. The ACM Transactions on Information and System Security **3** (2000)
5. Gray, J., Reuter, A.: Transaction Processing: concepts and techniques. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA (1993)
6. Ribeiro, C., Zúquete, A., Ferreira, P., Guedes, P.: Spl: An access control language for security policies with complex constraints. In: Network and Distributed System Security Symposium (NDSS'01), San Diego, California (2001)
7. Denning, D.: A lattice model of secure information flow. Comm. of ACM **20** (1977)
8. Edwards, W.K.: Policies and roles in collaborative applications. In: ACM 1996 Conference on Computer Supported Work, New York, ACM Press (1996) 11–20

[3] All measurements were taken on a personal computer with a Pentium II at 333MHz running the Sun Java 1.2.2 virtual machine over Windows NT 4.0.