An Artificial Manager for Security Policies in Organizations

Karen García-Gamboa[1] and Raúl Monroy[1] and Jesús Vázquez[2]

[1]Tecnológico de Monterrey, Campus Estado de México,
Km 3.5 Carretera al Lago de Guadalupe,
Col. Margarita Maza de Juárez, Atizapán de Zaragoza,
Estado de México, Mexico, 52926
Tel: +52 55 5864 5751
`{A00472043,raulm}@itesm.mx`
[2]Banco de México
Avenida 5 de Mayo, Col. Centro, Del. Cuauhtémoc
México D.F., Mexico, 06059
`jjvazquez@banxico.org.mx`

Abstract. Security policies are rules or conventions aimed at protecting the resources of an organization. Designing, implementing and maintaining security policies are all difficult error prone and time consuming tasks. We report on an e-policy manager, capable of expressing and reasoning about security policies, using (a subset of) first-order logic and closely following the work of [1]. The tool includes an interface that provides a graphical description of a set of policies. It also includes a link to an automated theorem prover, Otter [2], which is used to formally verify that the policies are consistent one another.

1 Introduction

Every organization should declare, and then adopt, a collection of security policies, for its resources to be protected. A *security policy* is a rule or convention that prescribes how to use a resource, keeping it away from an attack. If properly defined, these policies help ensuring the goals of security, namely: integrity, confidentiality, and availability of information. They may also help other purposes of the organization, including legal, regulation and contractual. Security policies should be applicable and easy to understand. A security policy should be like a program specification: it details what is to be done but not how.

Writing security policies is an error-prone task: policies are often ambiguous, they clash one another or they are simply outdated. Writing security policies is also very time-consuming, since it amounts to developing general plans that guarantee that the organization site will exhibit the intended behavior, even though it is under attack. This situation prompts the construction of a tool that can help a user designing and developing proper security policies.

This paper reports on one such a tool, we call *e-policy manager*, portraying the following features:

1. E-policy manager eases the capture of a security policy, while guaranteeing it conveys when someone is (not) allowed to carry out an action.
2. E-policy manager formalizes each security policy, using Halpern and Weissman first-order logic formulation [1], and then proves they are not inconsistent using the theorem-prover Otter [2].

E-policy manager deals only with security policies concerned with the protection of information files.

Paper overview

The rest of this paper is organized as follows: In Section 2 we describe how to express and reason about security policies using first order logic. We also give a toy example set of a set of inconsistent security policies and present how Otter [2] is used to pinpoint inconsistency. In Section 3 we present a graphical user interface for easily capturing security policies. In Section 4 we show results from a psychological validity test carried on the E-policy manager. In Sections 5 and 6 we respectively show related work and the conclusions drawn from our research.

2 Reasoning about Security Policies Using FOL

Formal methods advocate the use of techniques strongly based in Mathematics, with which we can formalize informally presented problems and provide rigorous arguments about their properties. To express and reason about security policies, we use (a subset of) first-order logic with equality over a vocabulary, as it has proven to be enough for this purpose [1].[1]

2.1 Permitting and Denying Policies

In this paper, security policies are either *permitting* (respectively *denying*) or *contextual*. A permitting (respectively denying) security policy conveys the *conditions* under which someone, the *subject*, is allowed (respectively forbidden) to perform an *action* on some *object*. Accordingly, the vocabulary is assumed to contain at least four collection of predicate relations, one denoting subjects (agents, processes, officers, etc.), other denoting objects (files, databases, applications, etc.), other denoting actions (read, write, execute, etc.), and other denoting constraints (time, roles, etc.)

A security policy therefore states a relation between a *subject* (users), an *object* (information files, databases, etc.) and an *action* (read, write, modify, etc.). The action is limited generally by the use of some access mechanism (a security mechanism). More formally, a (permitting) policy is a sentence of the form:

$$\forall X_1{:}T_1, \ldots, X_n{:}T_n.\ C \rightarrow [\neg]permitted(S, A) \qquad (1)$$

[1] The reader is assumed to be familiar with the syntax and the semantics of First-Order Logic (FOL).

where C is a first-order formula, S and A are terms that, when valuated, return a subject and an action over some object respectively, and where $[\neg]$ indicates that $\neg$ may or may not appear in the formula. If $\neg$ does not appear in the formula, then the security policy is of type permitting. Throughout this paper, we shall respectively use $\forall X{:}T.\ P(X)$ and $\exists X{:}T.\ P(X)$ as an abbreviation of $\forall X.\ (T(\mathrm{X}) \rightarrow P(X))$, read "all T's are P's" and $\exists X.\ (T(X) \wedge P(X))$, read as "some T's are P's".

Notice that we need some structure in order to be able to reason about the object upon which an action is carried out. To that purpose, we use function symbols. For example, we use *read*(f) to mean "read from file f". These are the function symbols of most frequent use:

- *read*(*f*) to mean "read from file f";
- *write*(*f*) to mean "write onto file f";
- *modify*(*f*) to mean "modify file f";
- *delete*(*f*) to mean "delete file f"; and
- *create*(*f*) to mean "create file f".

2.2 Contextual Policies

A contextual policy is any first-order formula which states a specific view as to what counts as security within an organization. Four security properties may be included as an contextual security policy: i) integrity, ii) confidentiality, iii) availability and iv) authentication. Symbolically, these kinds of security policies do not change and so can be included or taken out at any time. Other contextual policies include *unicity*. For example, the predicate relating staff members and passwords is so that:

$$\forall X{:}\mathit{staff}.\ \exists Y{:}\mathit{passw}.\ (\mathit{passw\_of}(X, Y) \wedge \forall Z{:}\mathit{passw}.\ (\mathit{passw\_of}(X, Z) \rightarrow Y = Z))$$

2.3 The Environment

Called the *environment* [1], facts about the application domain, including properties of relations, are also captured using first-order logic. Given that E-policy manager was specifically designed to capture security policies of an organization, the environment comprehends relations for specifying objects such as department names, personnel members, etc. Employees, for instance, are all given a role and an affiliation, etc. For example, we write:

$$\mathit{role\_of}(X, Y)$$

$$\mathit{affiliated2}(X, Y)$$

to respectively mean "employee X, of type *staff*, has position Y, of type *position*" and "employee X, of type *staff*, belongs to department Y, of type *department*". Information has an owner and it is classified in terms of its relevance to the organization. For example, to respectively mean "information X, of type *info*, belongs to department Y, of type *department*" "information X, of type *info*, is in class Y, of type *class*" in symbols we write:

$$belongs2(X, Y)$$

$$of\_class(\mathrm{X}, \mathrm{Y})$$

Once completely captured, both the security policies and the environment, the context is all given to an automated reasoner. We have chosen to use Otter [2], since it is well-established both in industry and in academia. Using Otter, we verify that the model is not logically inconsistent, as well as querying the tool for specific capabilities of a user within the organization at hand.

It is well-known that first-order logic is semi-decidable. In our experiments, Otter was able to quickly find an inconsistency in the input security policies, (deriving the empty clause), if there was any, but usually spent a while, otherwise. Notice that, if the input security policies are not inconsistent, then Otter may run forever. Hence, sometimes we might not be able to find an argument supporting (respectively refuting) that the policies are consistent. To get around this problem, we would have to simultaneously apply Mace to the input security policies (see Section 5).

When asked a specific query about what a specific user can do, Otter also replied quickly. Our experiments confirm the theoretical results of [1]—our security policies are standard and not bipolar in their sense, and contain no inequalities in their antecedent—. If necessary, Otter can be manually configured so that a user (a security officer in this case) can select what methods should be applied in the search for the empty clause. If it derives the empty clause, Otter prints out a proof, which we use to automatically hint the user which security policies are thought to be in conflict.

Otter has been made to run in automatic mode, applying binary resolution, unary resolution or UR-Resolution, hyper-resolution, and binary paramodulation. Otter works searching for the empty clause, which in this case is an evidence of contradiction amongst the security policies.

2.3 An Example Set of Inconsistent Security Policies

In this section, we give a toy example of a set of inconsistent security policies and show how Otter is used to pinpoint inconsistency as well as a possible root of failure. The example is meant to illustrate that, since she is writing a number of policies, the security officer may not be aware of inconsistencies in a large document. Let us consider that the user introduces the following security policies:

1. Members of the IT department may read the information that belongs to that department, in symbols:

$$\forall X{:}staff,\ Y{:}info.\ (affiliated2(X, it) \wedge belongs2(Y, it) \rightarrow permitted(X, read(Y))).$$

2. Members of the IT department cannot read the security procedures.

$$\forall X{:}staff.\ (affiliated2(X, it) \rightarrow \neg permitted(X, read(security\_procedure))).$$

3. Security procedures are part of the IT department documents.

$$\rightarrow belongs2(securiy\_procedures, it).$$

Once formalized, these policies are given to Otter for a consistency check. We conclude inconsistency, given that Otter finds the empty clause:

```
------------------------------------------ PROOF -------------------------------------------
 1 []   x:staff    |   y:info
 2[]    -affiliated2(x,it)    | -belongs2(y,it)    | permitted(x,read(y)).
 3 []   x:info     | -affiliated2(x,it)  | -permitted(x,read(security_procedures)).
 4 []   x:info  |  -belongs2(x,it)       |  belongs2(security_procedures,it).
 -----> EMPTY CLAUSE at   0.25 sec ----> 4  [hyper, 1, 2, 3, 4]   $F. ---------------
---------------------------------------- end of proof -----------------------------------------
```

The example set of security policies is inconsistent, because policies 1 and 2 contradict one another. Policy 1 states that the staff members of IT are all allowed to read the IT information, while policy 2 states that they cannot read security procedures, which is IT information.

3 A Graphical User Interface

Writing, developing and maintaining security policies are all responsibilities of a security officer, who can not be assumed to be acquainted with formal methods. Formal methods require both significant skill and time (and therefore money) to use. To get around this problem, E-policy manager comes with a graphical user interface (GUI) that makes it easy to capture security policies by means of wizards and other graphical techniques. Also the GUI makes it easy both to correctly interpret the input security policies and to formalize them in FOL.

Through the GUI, a security officer can capture a set of security policies at a high, abstract level, by means of schemata. Although schemata restrict the expressiveness of our policy language, the output security policies contain the necessary ingredients for guaranteeing a simple but correct translation into our first-order language. Moreover, the GUI hides formal methods and knowledge representation out of the user, who no longer needs to be acquainted with these techniques.

3.1 Capturing a Permitting Security Policy

Given that a policy specifies the conditions under which a subject is allowed (respectively forbidden) to perform an action on some object, the GUI portrays a schemata, based on wizards, through which the user conveys several pieces of information. This involves the subject, e.g. users, processes, etc., the action, e.g. reading, writing, creating, etc., the object, an information file, the restrictions under which the action is permitted, e.g. a user role, a user affiliation, etc., and whether the action should be denied or permitted. The subject of a security policy can be a specific individual or a group of individuals that are related by some condition.

Figure 1 is a pictorial description of our security policy capturing schema. Users are allowed to write security policies giving objects the central role in the grammatical construction. Then, objects appear first, as illustrated in Figure 2.

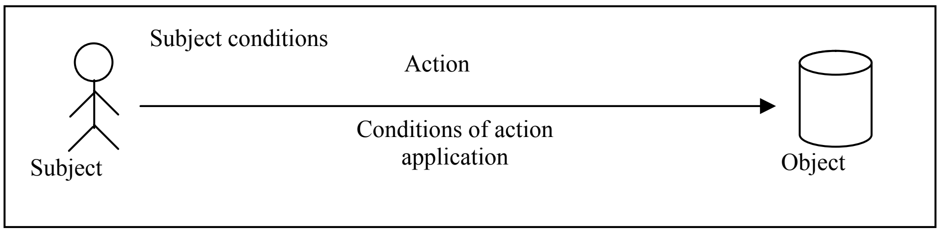

Figure 1. Security policy components

The GUI also allows the introduction of a modifier, we call the *purpose modifier*. If the purpose modifier is on, the interpretation of the security policy at hand is changed so that it now reads "*only* subject is allowed (respectively forbidden) to perform the associated action on the object under the given conditions".

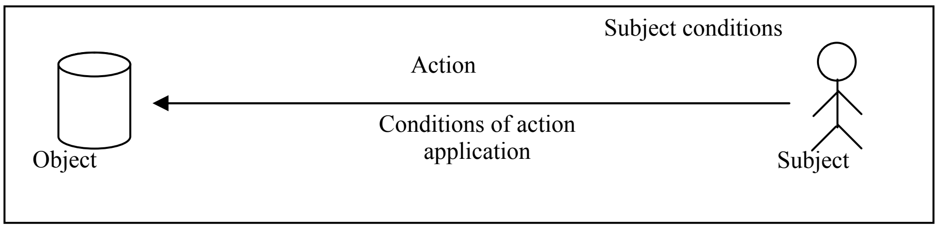

Figure 2. Security policies components (a different structure)

Using the GUI, a user is thus able to capture a security policy through a suitable schema. Schematically, the wizard enables the user to convey six pieces of information: i) the subject of the policy, ii) number of purposes (use only to denote exclusiveness, otherwise this field is left apart), iii) the type of policy (permitting or denying), iv) the action to be carried out, v) the object the action should be performed on, and vi) the constraints.

As an example policy capturing, consider that, after interacting with the wizard, the user has input the following information: `subject` =*staff*, `condition on the subject` = *officer*, `purpose modifier` = *on*, `policy type` = *permitting*, `action` = *read*, `object` = *passwords* (the password file), and `object constraints` = none, then the policy manager records the following formulae within the policy database:

$$\forall X{:}staff.\ (officer(X) \rightarrow permitted(X, read(passwords)))$$

$$\forall X{:}staff.\ (\neg officer(X) \rightarrow \neg permitted(X, read(passwords)))$$

As another example of security policy capturing, consider that, after interacting with the wizard, a security officer has input the following information: `subject` =

staff, `subject constraints` = *affiliated2*(*subject*, *it*), `purpose modifier` = off, `policy type` = *permitting*, `object` = *information*, `object constraint` = *belongs2*(*object*, *it*), `action` = *read*. Then, E-policy manager outputs:

$$\forall X{:}staff,\ Y{:}info.\ (affiliated2(X, it) \wedge belongs2(Y, it) \rightarrow permitted(X, read(Y)))$$

Each policy can be added or removed from the database by means of a wizard, which pops up a table containing all existing policies. A user only has to select an unwanted policy, by clicking on it, and then indicate policy removal or edition.

3.2 Linking the Environment with a Relational Database

The graphical user interface also drives a relational database. The user, making use of wizards, captures all the information related to the actual environment. For the application in mind, E-policy manager comes equipped with the following tables:

- Subject<identifier, name, department of affiliation, position>
- Object<identifier, name, department this object belongs to, class>
- Action<identifier, action description>
- Security_mechanism<identifier, access mechanism>

The relational database goal is not representing a security policy but providing the concrete information upon which the set policies act. Pretty much the same way in which policies are manipulated, the information hold in the database tables can be added, deleted, etc., via E-policy manager. When capturing a security policy, the schema (subject, action and object) can be filled with information retrieved from this relational database.

To provide flexibility to the user, we maintain a database of action synonymous and so a user may write (or select) “change”, “manipulate”, “alter”, etc. rather than the default “access”. To avoid slowing down the deduction process, we normalize all the synonymous of an action to a designated, default action name. This way, we do not perform additional, unnecessary applications of paramodulation or rewriting.

Using the capturing schema, each security policy is translated into both a first order formula and a semi-natural language expression. The formula is regarded as the formal model of the associated security policy. It is sent to Otter, if a consistency checking is to be performed. The expression is used for documentation purposes. It is part of the security policies manual of the associated organization.

The design of both the interface and the security policy schemata of E-policy manager were largely inspired in LaSCO [3]. LaSCO is an object oriented programming language which expresses a security policy by means of a constraint imposed on an object. Other policy languages were also considered, e.g. [4], but none of them provides as much a solid theoretical foundation as that of [1]. We have more to say about related work in section 5.

4 Testing Psychological Validity

E-policy manager was used to capture a number of security policies found in books or gathered from practitioners. Although they impose severe constraints on the policy language, schemata were found to be enough to capture all these security policies. Our experiments show that inconsistency checking may take a few milliseconds, if the security policies are contradictory, but may take a few seconds, otherwise. It is a theoretical result that FOL is undecidable; thus, in principle, inconsistency checking may not terminate at all.

E-policy manager was also evaluated for psychological validity. We run a test on six security officers, who answered a survey and interacted with the tool prototype. Our results from this experiment are encouraging. The answers provided by these security officers point that E-policy manager has achieved its two primary design goals, namely: i) to provide an graphical user interface that makes it easy to capture a security policy while guaranteeing it is correct in the sense of interpretation, and ii) to provide a means for formally verifying the security policies are consistent. Rather than an adverse opinion, we got words urging us to include an account for policies about other resources.

E-policy manager is available upon request by sending e-mail to the first author. It can also be downloaded from our URL, which we do not disclose here due to paper blind reviewing constraints. In the next section, we will review existing languages for the specification of security policies.

5 Related Work

A policy specification language aims at formalizing the intent of a policy designer into a form that can be read and interpreted by both people and machines [5]. It is especially designed to specify the relations amongst system entities in terms of actions and the conditions upon which these actions are denied or performed. There exist several policy specification languages. In what follows, we review the main features of 4 policy specification tools and associated languages: i) Keynote, ii) SPSL [5], iii) LaSCO [3], and the General Computer Security Policy Model [6].

Keynote and SPSL [5] are used to specify security policies about network applications. Neither Keynote nor SPL provide a visual tool for policy capturing. However, they are both equipped with a policy compiler, which produces a user profile that the intended application can use for denying or permitting the execution of an action. Keynote cannot be used to specify facts about the environment. Keynote does not scale properly, as it is difficult to foresee the state that results when enforcing a number of security policies. E-policy manager can be used to capture facts about the environment but was never thought as a mechanism for enforcing security policies.

LaSCO [3] is based on a model where a system consists of objects and events and works by conveying restrictions on objects. This language represents the policies by means of directed graphs which describe a specific state of the system (domain) and specific access constraints (requirements) and in mathematic logic. The nodes represent system objects and the edges represent system events. LaSCO [3] can be used to express a wide variety of standard and customized security policies, including access control and other history-based and context-dependent policies. Our work has been inspired in this language. For example, for the graphical user interface, we have adopted the use of graphs facilitating the security policies representation, as well as denoting information access control. LaSCO expressions can be translated into a lowlevel language for security policy enforcement. However, the tool does not involve the use of a mechanism to guarantee that the policies are consistent or that they meet certain properties.

Krsul, Spafford and Tuglualar [6] have presented a functional approach to the specification of security policies that allows policy stepwise refinement. The model makes the explicit assumption that policies and the value of the system objects are related. This model expresses policies as algorithmic and mathematical expressions. The specification policy explicitly lists the objects and attributes that are needed to enforce the policy. The model helps identifying the components that are relevant to the policy and hence provides a better policy understanding.

These languages are all adequate for the specification of security policies. However, they are not this effective, since, except for [6], they do not have a formal semantics, with which to reason about the security policies. Also security policy capturing using no visual aid has proven to be error prone, making it necessary to verify the written policies.

Halpern and Weissman have shown that (a subset of) first order logic is enough to express and efficiently reason about security policies [1]. They represent a security policy as a relation between three sorts, *Actions* (e.g. accessing a file), *Subjects* (the agents that perform actions) and *Times*. This contrasts with our work, where we denote a policy as a relation amongst *Subjects*, *Objects* and *Actions*. Halpern and Weissman are much more interested in using a user profile in order to enforce security policies; they argue that their security policy schema (which we have borrowed for our work) makes it easier for a user to write proper policies. They have not paid attention to checking policy consistency.

As we can see, our work is also based on Halpern and Weissman's. Indeed, in [1] it is mentioned these two researchers are working on developing a nice interface for security policy capturing, but no report has been published yet.

6 Conclusion and Future Work

To secure the most significant resources of an organization, it is necessary to have a set of appropriate policies. Managing security policies is not an easy task and currently it does not have computer support. The goal of our work is to provide a tool that supports this task and gives the bases for future research. The tool, called E-Policy manager, includes a graphical user interface that makes it easy to capture security policies and a module that formalizes these policies to be verified by a FOL automated theorem prover, Otter.

Our work in using FOL to formally represent and reason about security policies leaves plenty of room for improvement.

Further work includes simultaneously using Mace [8] and Otter, so as to obtain a conclusive result that takes into account the case of a consistent policy set (possibly making Otter run forever). Mace is a model generator which is usually used for searching a counterexample to an input conjecture. For our case, the counterexample will be (a subset of) the environment satisfying the input security policies.

Further work also includes using a natural language processor so as to allow a user to input security policies as he would in an informal document. This interface would significantly increment the acceptance of E-Policy manager from potential users.

References

1. Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop*, Pacific Grove, 2003. IEEE Computer Society.

2. William McCune. Otter 2.0 In Mark E. Stickel, editor, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 663-664. Springer, 1990.

3. James A. Hoagland. Specifying and enforcing policies using LaSCO, the language for security constraints on objects. *The Computing Research Repository*, cs. CR/0003066, 2000.

4. Renato Iannella. ODRL: The open digital rights language initiative. Technical report.

5. Minna Kangasluoma. Policy Specification Languages. Department of Computer Science, Helsinki University of Technology, 1999.

6. Ivan Krsul, Eugene Spafford, and Tugkan Tuglular. A New Approach to the Specification of General Computer Security Policies. COAST *Techical Report 97-13.*, 1998, West Lafayette, IN 47907–1398.

7. Davies E., *Representation of Commonsense Knowledge*, Courant Institute for Mathematical Sciences, 1990.

8. William McCune. Mace 2.0 Reference Manual and Guide, *Mathematics and Computer Science Division Techical Memorandum No. 249*. Springer, 2001.