Policy-Driven Negotiations and Explanations: Exploiting Logic-Programming for Trust Management, Privacy & Security

Piero A. Bonatti[1,⋆], Juri L. De Coi[2], Daniel Olmedilla[2], and Luigi Sauro[1]

[1] Università di Napoli Federico II
[2] L3S Research Center & University of Hannover

Abstract. Traditional protection mechanisms rely on the characterization of requesters by identity. This is adequate in a closed system with a known set of users but it is not feasible in open environments such as the Web, where parties may get in touch without being previously known to each other. In such cases policy-driven negotiation protocols have emerged as a possible solution to enforce security on future web applications. Along with this setting, we illustrate PROTUNE a system for specifying and cooperatively enforcing security and privacy policies (as well as other kinds of policies). PROTUNE relies on logic programming for representing policies and for reasoning with and about them.

1 Introduction

Open distributed environments such as the World Wide Web offer easy sharing of information, but provide few options for the protection of sensitive information and other sensitive resources. Even though latest developments such as the Web 2.0 have demonstrated that many users are willing to participate and therefore share information publicly, recent experiences with Facebook's "beacon" service[1] and Virgin's use of Flickr pictures[2] have also shown that users are not willing to accept every possible use (or abuse) of their data. Therefore, protection of services and sensitive data may determine the success or failure of a new service.

Policies with well-defined meaning and their exchange between parties during transactions allow for the dynamic enforcement of security and privacy. However, such a policy-aware web would equally fail if administrators and users do not understand such policies (their own and the ones from other parties they are interacting with), nor are they well informed about the process of enforcing them. Furthermore, in case a negotiation fails, receiving a simple "Transaction failed" is not satisfactory for a common user as it does not provide any clue about what has gone wrong.

⋆ In alphabetical order.

[1] http://www.washingtonpost.com/wp-dyn/content/article/2007/11/29/AR2007112902503.html?hpid=topnews

[2] http://www.smh.com.au/news/technology/virgin-sued-for-using-teens-photo/2007/09/21/1189881735928.html

M. Garcia de la Banda and E. Pontelli (Eds.): ICLP 2008, LNCS 5366, pp. 779–784, 2008.

In this paper, we present the PRovisional TrUst NEgotiation framework PROTUNE [1] which aims at combining distributed trust management policies with provisional-style business rules and access-control related actions. PROTUNE's rule language extends two previous languages: PAPL [2] and PEERTRUST [3], that supports distributed credentials and a flexible policy protection mechanism.

PROTUNE provides a framework with:

- a trust management language supporting (possibly user-defined) actions
- an extensible declarative metalanguage for driving decisions about information disclosure
- a parameterized negotiation procedure, that gives a semantics to the metalanguage and provably satisfies some desirable properties for all possible metapolicies
- general ontology-based techniques for smoothly integrating language extensions
- advanced policy explanations in order to answer why, why-not, how-to, and what-if queries [4]

Policies are basically sets of Horn rules (enhanced with some syntactic sugar) on which the system has to perform several kinds of symbolic manipulations such as deduction, abduction, and filtering (as described in Section 4).

We made use of both a tabled logic programming engine (XSB) and a Prolog compiled on Java bytecode. These two technologies have complementary advantages. Tabling significantly enhances performance in many cases, by factorizing common subproofs; moreover, tabling makes it possible to process recursive policies without worrying about termination. A direct implementation of the same features with procedural programming paradigms would raise implementation, debugging, and maintenance costs enough to prevent the adoption of similar enhancements. Java-based Prologs, on the other hand, facilitate deployment and on-the-fly code download by means of technologies that nowadays are installed on every computer (and well integrated with the security facilities of their browsers).

The following sections describe more in detail some of these features and how logic programming plays a crucial role in their definition and/or implementation.

2 Policy Specification

The PROTUNE rule language [1] is based on normal logic program rules "$A \leftarrow L_1, \ldots, L_n$" where A is a standard logical atom (called the *head* of the rule) and $L_1, \ldots, L_n$ (the *body* of the rule) are literals, that is, L_i equals either B_i or $\neg B_i$, for some logical atom B_i. In addition, PROTUNE is enhanced with a FLORA-like object oriented syntax that, however, is only an abbreviation for standard first-order syntax. One can express by $X.\texttt{attr} : v$ the fact that X has an attribute `attr` with value v. Actually, $X.\texttt{attr} : v$ abbreviates the standard atom $\texttt{attr}(X, v)$. This representation allows multi-valued attributes. This attribute semantics is compatible with semantic web standards such as RDF and OWL (in particular $X.\texttt{attr} : v$ corresponds to an RDF triple).

A *policy* is a set of rules, such that negation is applied neither to *provisional predicates* (defined below), nor to any predicate occurring in a rule head. This restriction ensures that policies are *monotonic* in the sense of [2], that is, as more credentials are released and more actions executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [5].

The vocabulary of predicates occurring in the rules is partitioned into the following categories.

- *Decision Predicates*: Currently supported are `allow/1`, which is queried for access control decisions, and `sign/1`, which is used to issue statements signed by the principal owning the policy.
- *Logical Predicates*: Comprise abbreviation and state-query predicates as described in [6]
- *Constraint Predicates*:Comprise the usual equality and disequality predicates
- *Provisional Predicates*: May be made true by executing associated actions that may modify the current state like e.g. `sentCredential/1`, `logged/2`, `sentDeclaration/1`.

3 Metapolicies

Metapolicies consist of rules similar to object-level rules. They allow to inspect terms, check groundness, call an object-level goal G against the current state (using a predicate $holds(G)$), etc. In addition, a set of reserved attributes associated to predicates, literals and rules (e.g., whether a policy is public or sensitive) is used to drive the negotiator's decisions. For example, if p is a predicate, then p.`sensitivity` : `private` means that the extension of the predicate is private and should not be disclosed. An assertion p.`type` : `provisional` declares p to be a provisional predicate; then p can be attached to the corresponding action α by asserting p.`action` :α. If the action is to be executed locally, then we assert p.`actor` : `self`, otherwise assert p.`actor` : `peer`.

As pointed out before, metarules and metaattributes may be used to attach provisional predicates to the corresponding actions. The language for local actions should be flexible and powerful, to facilitate the integration of trust management in the surrounding environment. Script languages are good candidates; multiple action languages may coexist in the same policy.

As an example, the predicate `logged` (which stores a message in a file) can be associated to its action by a simple metafact or an ontology definition:

`logged`(Msg, $File$).`action`: $'$`echo`$'$ + Msg + $'>'$ + $File$.
`logged`(Msg, $File$).`ontology`:< `www.L3S.de/policyFramework#Logged` > .

The exit status of the action determines whether the corresponding provisional atom is asserted.

4 Negotiations, Policy Reasoning and Filtering

In open distributed environments like the World Wide Web parties may make connections and interact without being previously known to each other. Therefore, before any meaningful interaction starts, a certain level of trust must be established from scratch. Generally, trust is established through exchange of information between the two parties. Since neither party is known to the other, this trust establishment process should be bi-directional: both parties may have sensitive information that they are reluctant to disclose until the other party has proved to be trustworthy at a certain level. This process is called trust negotiation and, if every party defines its access control and release policies to control outsiders' access to their sensitive resources, can be automated.

Therefore, during a negotiation both a requester (client) and a server exchange their policies and information with the goal of performing a transaction. The use of set of horn rules for policies together with ontologies provide the advantage of well-defined semantics and machine interoperability, hence allowing for automated negotiations. When a set of policy rules is disclosed by a server in response to a client's request, the client—roughly speaking—works back from the request (goal) looking for the provisional predicates such as credentials and declarations in its portfolio that match the conditions listed in the rules' bodies. In logical terms, the selected credentials and declarations (represented as logical atoms) plus the policy rules should entail the goal: this is called an *abduction problem*. After receiving credential and declarations from a client, a server checks whether its policy is fulfilled by trying to prove the goal using its own rules and the new atoms received from the client, as in a standard *deduction problem*. When a client enforces a privacy policy and issues a counter-request, the roles of the two peers are inverted: the client plays the role of the server and viceversa.

However, policies (or parts thereof) may be sensitive as well, and therefore they should not be exchanged unless there exists enough level of trust on the other party. PROTUNE provides a filtering mechanism [1] which taking into account specified metapolicies (e.g., sensitivity, actor and execution) as well as the information received from the other party may partially hide the "filtered" policy to be sent to the other party

5 Explanations

The frameworks for protecting security and privacy can be effective only if common users—with no training in computer science or logic—increase their awareness and control over the policy applied by the systems they interact with. Towards this end, PROTUNE introduces a mechanism for answering *why, why-not, how-to*, and *what-if* queries on rule-based policies [4], using simple generic explanation strategies based on the intended meaning of a few core predicates with a special role in negotiations. PROTUNE is *lightweight* and *scalable*. The

only extra workload needed during the framework instantiation phase to support explanations consists in writing literal verbalization patterns. Moreover, the extra computational burden on the server can be limited to adding a few more rules to the filtered policies (the literal verbalization rules) because the explanations can be independently produced on the clients. Despite its simplicity, our explanation mechanism supports most of the advanced features of second generation explanation systems. Moreover, it adopts a novel *tabled explanation structure*, that simultaneously shows local and global (intra-proof and inter-proof) information, thereby facilitating navigation. To focus answers in the trust negotiation domain, suitable heuristics are introduced in order to remove the irrelevant parts of the derivations. There are several novel aspects in such an approach:

- We adopt a *tabled explanation structure* as opposed to more traditional approaches based on single proof trees. The tabled approach makes it possible to describe infinite failures (which is essential for *why not* queries).
- Our explanations show the outcome of different possible proof attempts and let users see both local and global proof details at the same time. Such combination of intra-proof and inter-proof information is expected to facilitate navigation across the explanation structures.
- We introduce suitable heuristics for focussing explanations by removing irrelevant parts of the proof attempts. Anyway, we provide a second level of explanations where all the missing details can be recovered, if desired.
- Our heuristics are *generic*, i.e. domain independent. This means that they require no manual configuration.

6 Implementation: The Protune Framework

In PROTUNE policies are basically sets of Horn rules (enhanced with some syntactic sugar) on which the system has to perform several kinds of symbolic manipulations such as deduction, abduction, and filtering. All these forms of reasoning can be implemented with suitable metainterpreters; logic programming languages are perfect for these purposes. Features such as tabling (when available) lead to significant performance improvements with no extra implementation costs—this would not be conceivable with any other programming paradigms. Moreover, we make extensive use of a logic programming language compiled onto Java bytecode as part of a strategy for simplifying the deployment of our user agents, that can be even downoladed dynamically as applets. The rest of the framework and extensible interfaces are provided in Java in order to improve portability. A live demo of PROTUNE in a Web scenario is publicly available[3] as well as a screencast[4]. For a demo of the explanation facility and extensive documentation see `http://people.na.infn.it/rewerse/`

[3] `http://policy.l3s.uni-hannover.de/`

[4] `http://www.viddler.com/olmedilla/videos/1/`. Recommended in full screen.

References

1. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: IEEE POLICY, Stockholm, Sweden (2005)
2. Seamons, K., Winslett, M., Yu, T., Smith, B., Child, E., Jacobsen, J., Mills, H., Yu, L.: Requirements for Policy Languages for Trust Negotiation. In: IEEE POLICY, Monterey, CA (2002)
3. Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K.E., Winslett, M.: No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In: Bussler, C.J., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053. Springer, Heidelberg (2004)
4. Bonatti, P.A., Olmedilla, D., Peer, J.: Advanced policy explanations on the web. In: 17th European Conference on Artificial Intelligence (ECAI 2006), Riva del Garda, Italy (2006)
5. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, Cambridge (2003)
6. Bonatti, P., Samarati, P.: Regulating Service Access and Information Release on the Web. In: ACM Conference on Computer and Communications Security, Athens (2000)