Protune: a framework for semantic web policies

P.A. Bonatti
L.Sauro
Università di Napoli Federico II

J.L. De Coi
D. Olmedilla
University of Hannover

Categories and Subject Descriptors
K.6.5 [Management of Computing and Information Systems]: Security and Protection

Keywords
Trust negotiation, Semantic Web Policies

1. THE PROTUNE VISION

Security and privacy play a central role in raising the level of trust in web resources. Web services obviously need some form of access control, whereas users are not willing to accept every possible use (or abuse) of their data. Therefore, the application of suitable policies for protecting services and sensitive data may determine success or failure of a new service.

In this paper we describe some features of the policy framework Protune, designed and implemented within the network of excellence REWERSE to support the security and privacy on the web. The challenges Protune tackles consist in two major issues: first Protune does not only enable traditional access control but also trust negotiation where both the policies of the two peers are taken into account in a request/counter-request process. Thus, it is possible to automate security checks and privacy-aware information release. Secondly, increasing protection should not affect the usability, therefore on one hand common users have not to be discouraged by the extra steps required by the trust negotiation, and on the other hand they have to be aware of which policy is actually applied by a system, which are its consequences and compare their own privacy requirements with whatever privacy policy is advertised by a web service.

A leading idea in designing Protune has been that policies are semantic markup because they specify declaratively part of the semantics (in terms of behavior constraints and admissible usage) of the static or dynamic resources that policies are attached to. Accordingly, semantic techniques have several roles in Protune:

- Policies are formulated as sets of axioms and meta-axioms with a formal, processing-independent semantics; this is the basis for consistent treatment of policies for different tasks: enforcement, negotiation, explanations, validation, etc.
- The aforementioned tasks involve different automated reasoning mechanisms, such as deduction for enforcement, abduction and partial evaluation for negotiation, pruning and natural language generation for explanations, etc.
- The auxiliary concepts needed to formulate policies (such as what is an accepted credit card, ...) and the link between such concepts and the evidence needed to prove their truth (e.g. which X.509 credentials are needed to prove authentication, or what forms need to be filled in) are defined by means of lightweight ontologies that may be included in the policy itself or referred to by means of suitable URIs; therefore, unlike XACML contexts, Protune’s auxiliary concepts are machine understandable and allow agent interoperability.

Here we give just a quick view of the Protune’s framework, more information can be found in the foundational works [1, 2] and on the web site of REWERSE’s working group on Policies: http://cs.na.infn.it/reverse/. There, on the software page, the interested reader may find links to Protune’s software and some online demos and videos.

2. THE PROTUNE FRAMEWORK

2.1 Policy language

Protune’s policy language is a logic programming language enhanced with an object oriented syntax. For example, the rule that allows to buy a book by giving a credit card could be encoded with a set of rules including:

allow(buy(Resource)) ←

credential(C), valid_credit_card(C), accepted_credit_card(C).

valid_credit_card(C) ←

C.expiration : Exp, date(Today), Exp > Today.

where C.expiration : Exp is an O.O. expression meaning that Exp is the value of C’s attribute expiration. Protune policies may use and define different categories of predicates:

Decision predicates are used to specify a policy’s outcome, such as allow() in the above example.

Provisional predicates are meant to represent actions performed during the evaluation of a policy. For example, one may want to retrieve the current system time, to send a query to a database or to record some information in a log file. In particular, Protune supports two pre-defined provisional predicates: credential and declaration. An atom credential(x) is true when an object x representing an X.509 credential is stored in the current negotiation state. A peer may make credential(x) true on the other peer by sending the corresponding credential; this is the action attached to this particular provisional predicate. Predicate declaration is analogous but its argument x is an unsigned semi-structured object similar to a web form that, for example, can be used to encode a traditional password-based authentication procedure.
Abbreviation predicates enable to define the auxiliary concepts needed to formulate policies (such as \texttt{valid\_credit\_card}) and the link between such concepts and the evidence needed to prove their truth, that is the conditions listed in rule bodies.

The properties regarding the predicates involved in the definition of a policy are defined by means on metarules. Metarules enable for example to specify a predicate as provisional, the corresponding action and the actor in charge of executing the action, for example:

\[
\begin{align*}
\log(X) & \rightarrow \text{type} : \text{provisional}. \\
\log(X) & \rightarrow \text{action} \neq \text{\texttt{echo \$X > logfile'}}. \\
\log(X) & \rightarrow \text{actor} : \text{self}.
\end{align*}
\]

where “\texttt{\textasciitilde}” connects a metaterm to its metaproperties.

Rules and ontologies may be sensitive. For example, in a social network scenario a rule such as

\[
\text{allow(download(pictures))} \leftarrow \text{best\_friend}
\]

may have to be protected, because in case of a denial it may reveal to a friend that he or she is not considered as a best friend. The sensitivity level of predicates and rules is also defined with metapolicies that drive a policy filtering process that selects relevant rules (for efficiency), and removes sensitive parts if needed. The first definition of policy filtering \cite{1} performed also partial evaluation w.r.t. the available facts. The current implementation does no partial evaluation anymore because (i) it may significantly increase the size of the messages exchanged during negotiations, and (ii) it destroys much of the structure of the policies thereby making the explanation facility (illustrated later on) much less effective.

2.2 The engine

Protune can be entirely compiled onto Java bytecode. Network communications and the main flow of control for negotiations are implemented directly in Java, while reasoning (including filtering) is implemented in tuProlog, a standard Prolog that can be compiled onto Java bytecode.

At each step of the trust negotiation process a peer \(P_1\) sends another peer \(P_2\) a (potentially empty) filtered policy and a (potentially empty) set of notifications, respectively stating the conditions to be fulfilled by \(P_2\), and notifying the execution by \(P_1\) of any actions it was asked for. As soon as \(P_2\) receives this information, it adds it to its negotiation state.

Then \(P_2\) processes its local policy in order to identify the local actions that can be performed taking into account the new information received. When such local actions are performed, other local actions may become ready for execution: this is the case e.g., if the instantiation of a variable is a prerequisite for the execution of an action and the instantiation of this variable is (part of) the result of another action’s execution. For this reason local action selection and execution are performed in a loop, until no more actions are ready to be executed. After having performed all possible local actions the local policy is processed in order to check whether the overall goal of the negotiation is fulfilled. If this is the case, a message is sent to \(P_1\) telling that the negotiation can be successfully terminated. Otherwise a termination criterion is used to decide whether the negotiation should continue or fail.

If the negotiation is not yet finished, it is \(P_2\)’s turn to filter its local policy and collect all items that have to be sent back to \(P_1\). Then, it processes its local policy and the (last) filtered policy received from \(P_1\) in order to select which of the actions whose execution has been requested by \(P_1\) will be performed. Notice that only actions such that the policies protecting them are fulfilled (\textit{unlocked actions}) might be actually performed.

2.3 Explanations: Protune-X

Protune-X, the explanation facility of Protune, plays an essential role in improving user awareness about—and possibly control over—the policy enforced by a system. Protune-X is also a major element of Protune’s cooperative enforcement strategy: Since a crude denial may discourage new users from using a system, the explanation system is meant to enrich the denials with information about how to obtain the permissions (if possible) for the requested service or resource.

For this purpose four kinds of queries are supported: \textit{How-to} queries provide a description of a policy and may help a user in identifying the prerequisites needed for fulfilling the policy. \textit{What-if} queries are meant to help users foresee the results of a hypothetical situation, which may be useful for validating a policy before its deployment. Finally, \textit{why} and \textit{why-not} queries explain the outcome of a concrete negotiation (i.e. provide a context-specific help). Some of the major desiderata guided the design of Protune-X:

\begin{itemize}
  \item Explanations should not increase significantly the computational load of the servers. For this reason explanations are produced in our approach by a distinct module, ProtuneX, which operates client-side.
  \item Almost no further effort should be added to the policy instantiation phase. This is achieved by exploiting generic heuristics as much as possible. For example how-to explanations exploit the \textit{actor} meta-attributes to distinguish automatically the prerequisites that should be satisfied by users from the conditions that are locally checked by the server. In most cases, the only extra effort needed for enabling explanations consists in writing verbalization metarules in order to specify how single, domain-specific atoms have to be rendered, e.g.:
    \[
    \text{password(X, Y)} \leftarrow \text{verbalization} : [Y, ‘is the password of’, X].
    \]
    \textit{Explanations should support so-called second generation features.} Such features include methods to present the explanation in manageable pieces, highlight relevant information while pruning irrelevant parts, present explanations in a user friendly fashion rather than following the engine’s artificial reasoning method.
\end{itemize}

3. REFERENCES

\begin{enumerate}
\end{enumerate}