Protune

January 21, 2007

1 Protune Negotiation Model

This document aims at formalizing the negotiation between two entities, namely E_1 and E_2. For the rest of the paper we assume that E_1 is the initial requester while E_2 is the provider, i.e. E_1 is assumed to send a request to E_2 thus starting a negotiation. Notice that during a negotiation both entities E_1 and E_2 may both request or provide information to each other.

Let our Policy language be a rule-based language. Such a rule language is based on normal logic program rules $A \leftarrow L_1, \dots L_n$ where A is a standard logical atom (called the *head* of the rule) and $L_1, \dots L_n$ (the *body* of the rule) are literals, i.e. L_i equals either B_i or $\neg B_i$, for some logical atom B_i.

Definition 1 (Policy) *A Policy is a set of rules, such that negation is not applied to any predicate occurring in a rule head.*

This restriction ensures that policies are *monotonic* in the sense of [**?**], i.e. 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 [**?**].

Definition 2 (Negotiation Message) *A Negotiation Message is an ordered pair*

$$(p, C)$$

where

- $p \equiv$ *a Policy*
- $C \equiv$ *a set of credentials*

We will denote with M the set of all possible Negotiation Messages.

Definition 3 (Message Sequence) *A Message Sequence σ is a list of Negotiation Messages*

$$\sigma_1, \dots \sigma_n \mid \sigma_i \in M$$

We will denote with $|\sigma|$ the length of σ and with σ_i the i-th element of the Message Sequence σ.

Definition 4 (Negotiation History) *Let E_1 and E_2 be the two entities involved in the negotiation. Let E_1 be the initiator of such a negotiation, i.e. the sender of the first message in the negotiation. A Negotiation History σ for the entity E_j $(j = 1, 2)$ is a Message Sequence*

$$\sigma_1, \dots \sigma_n \mid \sigma_i \in M$$

Moreover let

- $M_{snt}(\sigma) = \{\sigma_i \mid i = 2k - (j\ mod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$
- $M_{rvd}(\sigma) = \{\sigma_i \mid i = 2k - 1 + (j\ mod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$

We will refer to a Negotiation History also as a *Negotiation State.*

Intuitively, messages among parties are sent alternatively, i.e. a message sent by E_j is followed by the reception of a message, which is in turn followed by the sending of a new message and so on. Therefore, $M_{snt}(\sigma)$ (resp. $M_{rvd}(\sigma)$) represents the set of messages sent (resp. received) by E_j.

Notice that according to this definition, the Negotiation History σ is shared by the two entities E_1 and E_2, but the sets $M_{snt}(\sigma)$ and $M_{rvd}(\sigma)$ are swapped among them. Therefore it holds that

$$M_{snt}(E_1, \sigma) = M_{rvd}(E_2, \sigma)$$

and

$$M_{rvd}(E_1, \sigma) = M_{snt}(E_2, \sigma)$$

In order to ease the notation in the rest of the paper, given a Negotiation History σ, we define the following entities

- $C_{snt}(\sigma) = \bigcup\{C_i \mid \exists p_i\ (p_i, C_i) \in M_{snt}(\sigma)\}$
- $C_{rvd}(\sigma) = \bigcup\{C_i \mid \exists p_i\ (p_i, C_i) \in M_{rvd}(\sigma)\}$
- $lp_{snt}(\sigma) = p_{i_{max}} \mid i_{max} = max\{i \mid (p_i, c_i) \in M_{snt}(\sigma)\}$
- $lp_{rcv}(\sigma) = p_{i_{max}} \mid i_{max} = max\{i \mid (p_i, c_i) \in M_{rcv}(\sigma)\}$

Intuitively, C_{snt} (resp. C_{rvd}) represents the set of all credentials sent (resp. received) and lp_{snt} (resp. lp_{rvd}) represents the last policy sent (resp. received).

Definition 5 (Negotiation State Machine) *A Negotiation State Machine is a tuple*

$$(\Sigma, S, s_0, t)$$

such that

- $S \equiv$ *a set of Negotiation States*
- $s_0 \equiv$ *the empty list (initial state)*

- $\Sigma \equiv$ *a set of Negotiation Messages.*
- $t \equiv$ *a function* $S \times \Sigma \to S$ *such that if* $S = (\sigma_1, \ldots \sigma_n)$ *then* $t(S, \sigma) = (\sigma_1, \ldots \sigma_n, \sigma_{n+1})$ *(transition function)*

Intuitively a Negotiation State Machine models how an entity evolves during the negotiation by the exchange of messages. Σ contains both sent and received Negotiation Messages and can therefore be partitioned into two subsets Σ_{snd} and Σ_{rcv}.

Definition 6 (Negotiation Model) *A Negotiation Model is a tuple* (C, P, p_0, NSM, ff, ns) *where*

- $C \equiv$ *a set of credentials*
- $P \equiv$ *a set of Policies*
- $p_0 \equiv$ *a Policy (local Policy)*
- $NSM \equiv$ *a Negotiation State Machine* (Σ, S, s_0, t)
- $ff \equiv$ *a function* $S \to P$ *(Filtering Function)*
- $ns \equiv$ *an ordered pair* (csf, ta) *where*
  - $csf \equiv$ *a function* $S \to C$ *(Credential Selection Function)*
  - $ta \equiv$ *a function* $S \to \{true, false\}$ *(Termination Algorithm)*

  (Negotiation Strategy)

Each occurrence of S is supposed to refer to the same set of Negotiation States.

The intended meaning is as follows

- C represents the set of the credentials local to the Peer
- p_0 represents the Peer's policy protecting the local credentials and allowing access to the local resources
- S represents the set of states in which the Peer can be
- s_0 represents the initial state, i.e. the state in which the Peer is at the beginning of the negotiation
- f represents the process of filtering the Peer's Policy according to the current state
- csf represents the process of selecting the Peer's credentials to send to the other Peer
- ta represent the Peer's decision about whether going on or terminating the current negotiation

2 Protune to Prolog Rewriting Rules

Translation is needed between the Protune representation and the implementation languages used for the inference. Currently we have chosen Prolog as implementation language and the following are the translation rules used in order to transform Protune rules into Prolog rules.

The Protune to Prolog parser, implemented in JavaCC, receives as input a policy written in Protune. It processes each directive, rule and metarule contained in the policy given and then, the rules and the metarules are translated to a Prolog representation. Finally, it stores the directives and the translated rules and metarules in separate vectors.

General rules are transformed as follows:

$$[Id]Head \leftarrow L_1, \cdots, L_n.$$
$$\longrightarrow$$
$$rule(Id, Head, [L_1, \cdots, L_n]).$$

If the rule does not have a body then the third argument of the rule predicate, which is a list, is empty. Identifiers for rules written in Protune are optional, they can be specified or not. However during the filtering, it is essential that each rule has an identifier in order to distinguish among them while processing the policy. The translator is then responsible for assigning default different identifiers to each rule missing one.

Metarules are transformed as follows:

$$[Id].Attribute : Value \leftarrow L_1, \cdots, L_n.$$
$$\longrightarrow$$
$$metarule(id, Attribute(Id, Value), [L_1, \cdots, L_n]).$$
$$Head.Attribute : Value \leftarrow L_1, \cdots, L_n$$
$$\longrightarrow$$
$$metarule(pred, Attribute(Head, Value), [L_1, \cdots, L_n]).$$

If a metarule is without a body, then the third argument of the metarule predicate in the Prolog representation, which is a list, is empty. When translating metarules, two particles are introduces “id” and “pred” in order the mark that each metarule refers to a rule using the rule id or to a predicate using a literal which unifies with it. This is necessary as an id which is a simple string constant is identical with a predicate without any arguments, so it can be ambiguous.

Special care must be taken when translating complex terms which in general are transformed as follows:

$$Id[Attribute_1 : Value_1, \cdots, Attribute_n : Value_n]$$
$$\longrightarrow$$
$$complex\_term(Id, Attribute_1, Value_1), \cdots, complex\_term(Id, Attribute_n, Value_n).$$

However, the translation is different depending on the position of a rule where the complext term occurs:

- It is in the head of a rule:

$$
\begin{array}{l}
[RId]CId[attribute_1 : Value_1, \cdots, attribute_n : Value_n] \leftarrow L_1, \cdots, L_n. \\
\quad \longrightarrow \\
\qquad rule(RId, complex\_term(CId, attribute_1, Value_1), [L_1, \cdots, L_n]). \\
\qquad \cdots \\
\qquad rule(RId, complex\_term(CId, attribute_n, Value_n), [L_1, \cdots, L_n]).
\end{array}
$$

- It is an argument of a predicate that is a rule head:

$$
\begin{array}{l}
[RId]pred(CId[attribute_1 : Value_1, \cdots, attribute_n : Value_n]) \leftarrow L_1, \cdots, L_n. \\
\quad \longrightarrow \\
\qquad rule(RId, pred(CId), [L_1, \cdots, L_n]). \\
\qquad rule(RId, complex\_term(CId, attribute_1, Value_1), [L_1, \cdots, L_n]). \\
\qquad \cdots \\
\qquad rule(RId, complex\_term(CId, attribute_n, Value_n), [L_1, \cdots, L_n]).
\end{array}
$$

- It is in the body of a rule:

$$
\begin{array}{l}
[RId]pred() \leftarrow L_1, \cdots, CId[attribute_1 : Value_1, \cdots, attribute_n : Value_n], \cdots, L_n. \\
\quad \longrightarrow \\
\qquad rule(RId, pred(), [L_1, \cdots, complex\_term(CId, attribute_1, Value_1), \\
\qquad \cdots, complex\_term(CId, attribute_n, Value_n), \cdots, L_n]).
\end{array}
$$

- It is an argument of a predicate in the body of a rule

$$
\begin{array}{l}
[RId]pred_1() \leftarrow L_1, pred_2(\cdots, CId[attribute_1 : Value_1, \cdots, attribute_n : Value_n]), \cdots, L_n. \\
\quad \longrightarrow \\
\qquad rule(RId, pred_1(), [L_1, \cdots, pred_2(CId), complex\_term(CId, attribute_1, Value_1), \\
\qquad \cdots, complex\_term(CId, attribute_n, Value_n), \cdots, L_n]).
\end{array}
$$

Transformations for complex terms which appear in metarules are similar to the ones described above but complex terms cannot appear in the head.

In order to make all these translations possible when a complex term is discovered in the input stream, its elements are stored in a vector and this vector is passed along as part of the object returned by a method. For example, in case the complex term appears in a predicate in the body of the rule, the id of the complex term is stored as a predicate argument and the elements of the complex term are passed along and placed after the predicate. If the complex term appears in the head of a rule then the elements of the complex term need to be passed along until the entire rule is matched. In the case where more complex terms appear as arguments of the same predicate, then their elements are merged together in a single vector and then passed along.

Another intresting issue is translating special built-in predicates. The translation is performed as follows:

$$\begin{aligned}&in(pred(Arg^1_1,\cdots,Arg^1_n),package:function(Arg^2_1,\cdots,Arg^2_m)).\\&\quad\longrightarrow\\&\qquad in(pred(Arg^1_1,\cdots,Arg^1_n),package,function,[Arg^2_1,\cdots,Arg^2_m]).\end{aligned}$$

The arguments for the package call function are gathered in a list, so is be easier to integrate modules for making package calls.

The start method of the parser which must match the main BNF production, for the entire policy, is extended. This happens as in the policy rules and metarules can be mixed and have common tokens. Since identifying the start of a rule or a metarule is frequently done while translating, we preferred to factor together the rule and the metarule syntax instead of using local lookahead, in order not to slow down the transformation process. Local lookahead is also used in other situations were factorization of rules would have been difficult and would have made the grammar not welcome future changes very well.It must be mentioned as well that even though the rules referring to literal and metaliteral have common parts, they were implemented separately in order to favor factorization over local lookahead. An important feature of the translator is that while transforming the Protune policy in the Prolog representation the semantics of the policies is not changed, only the syntax is modified.

3 Semantics-preserving policy filtering

We parameterize policy filtering in order to be able to modify the filtering process using metadata. For the filtering techniques reported in this section, we shall prove that the choice of the filtering criteria does not affect correctness/completeness.

3.1 Removing Irrelevant Rules

This is an instance of the *need to know principle.* The *relevant subset* of a policy Pol w.r.t. an atom A is the least set S such that:

1. If the head of a rule $R \in Pol$ unifies with A, then $R \in S$;
2. If the head of a rule $R \in Pol$ unifies with an atom B occurring in the body of some rule in S, then $R \in S$.

The relevant subset of Pol w.r.t A will be denoted by

$$\mathsf{relevant}(Pol, A)\,.$$

The relevant subset of Pol w.r.t A suffices to determine which instances of A are entailed by the policy in the given state:

Lemma 1 *For all ground atoms $A\theta$ (θ is a substitution),*

$$A\theta \in \mathsf{cmodel}(Pol, \Sigma) \text{ iff } A\theta \in \mathsf{cmodel}(\mathsf{relevant}(Pol, A), \Sigma)\,.$$

3.2 Compiling Private Policies

The *immediate consequences* of a rule R w.r.t. *Pol* and Σ are the heads of the (ground) rules $R' \in \{R\}^{\Sigma}$ whose body is true in $\mathsf{cmodel}(Pol, \Sigma)$. The set of all immediate consequences of R w.r.t. *Pol* and Σ is denoted by $\mathsf{cons}(R, Pol, \Sigma)$. This operator is extended to policies in the natural way:

$$\mathsf{cons}(Pol', Pol, \Sigma) = \bigcup_{R \in Pol'} \mathsf{cons}(R, Pol, \Sigma).$$

Intuitively, cons *compiles* the subpolicy Pol' and replaces it with its immediate consequences. In this way, the results of the policy may be released to the peer without disclosing the internal structure of the rules.

This transformation preserves the semantics of the given policy, no matter what rules are compiled:

Theorem 1 *For all Pol, Pol′, Σ,* $\mathsf{cmodel}(Pol \cup Pol', \Sigma) =$

$$\mathsf{cmodel}(Pol \cup \mathsf{cons}(Pol', Pol \cup Pol', \varSigma), \Sigma)\,.$$

4 Filtering with information loss

Policies and states are sensitive resources. In general it may be necessary to hide part of them, which necessarily causes some information loss.

4.1 Blurring

Some rules R may have to be hidden and blocked until the client is trusted enough. This is accomplished by means of suitable metastatements:

$$\hat{R}.\mathtt{sensitivity} : \mathtt{not\_applicable} \leftarrow \ldots\,.$$

(where $\hat{R}$ is R's name). As more credentials arrive, R may become visible and extend negotiation opportunities. In this framework, policy disclosure has a reactive flavour, as opposed to the predefined graph structure adopted in [**?**].

Similarly, sensitive state predicates may have to be blocked until their evaluation does not disclose confidential information.

However, they cannot simply be left in the policy and sent to the client[1] because

- the client does not know how to evaluate them, since it has no access to the server's state, and
- the syntax of protected conditions may suffice to disclose some confidential information about the structure of the policy.

[1]Hereafter by "client" we mean the peer that submitted the last request, and by "server" we denote the peer that is evaluating its local policy to decide whether the request should be accepted and whether a counter-request is needed.

Removing these occurrences from the rules is not a good solution either, because then the client would not be aware that some conditions that lie beyond its control shall be checked later by the server. The client should be able to see that even if all credentials occurring in the policy were supplied, still the requested access might be denied. More precisely, the client should be able to distinguish the credential sets that satisfy the server's request with no additional checks, from the credential sets that are subject to further verification.

The solution adopted here consists in *blurring* the state conditions that cannot be evaluated immediately and cannot be made true by the other party. Such conditions are blurred by replacing them with a reserved propositional symbol.

For example, consider again the login policy (**??**). To avoid information leakage we postpone the evaluation of $\texttt{user}(\texttt{U},\texttt{P})$ and send the client a modified rule:

$$\begin{aligned}&\texttt{allow}(\texttt{enter\_site}()) \leftarrow\\ &\quad \texttt{declaration}(\texttt{usr}=\texttt{U},\texttt{passwd}=\texttt{P}),\texttt{blurred}\end{aligned}$$

where r is the name of rule (**??**). From this rule, a machine may realize that sending the declaration does not suffice to enter the site; first the server is performing a check of some sort.[2] Blurring is formalized below.

Let B be a set of literals, specifying which literals have to be blurred. For all rules $R = (A \leftarrow \mathit{Body})$ with name r, let $\mathsf{blur}(R,B) = (A \leftarrow \mathit{Body}')$ where

- $\mathit{Body}' = \mathit{Body}$ if $\mathit{Body} \cap B = \emptyset$, and
- $\mathit{Body}' = (\mathit{Body} \setminus B) \cup \{\texttt{blurred}\}$ otherwise.

Then for all policies Pol, define

$$\mathsf{blur}(\mathit{Pol},B) = \bigcup_{R\in \mathit{Pol}} \mathsf{blur}(R,B)\,.$$

To prove the effectiveness of blurring in protecting the internal state, we show that under suitable conditions, the blurred partial evaluation of any given policy Pol is invariant across all possible contents of the protected part of the state. As a consequence, from the result of the blurring one cannot deduce any protected state literal.

To formalize this, say two states are equivalent if they have the same non-blurred (public) part:

$$\Sigma \equiv_B \Sigma' \text{ iff } \Sigma \setminus B = \Sigma' \setminus B\,.$$

[2]Normally declarations result in a pop-up window where the user can directly type in the requested information or click an *accept* button. If the declaration is to be handled automatically, the client's policy should encode enough information to relate the appropriate user-password pair to the current service request. Moreover, appropriate policy rules are needed to decide whether the user should be queried or the declaration should be handled automatically.

Theorem 2 (Confidentiality) *For all Pol, Σ, Σ', E and B of the appropriate type, if $E \cap B = \emptyset$ and $\Sigma \equiv_B \Sigma'$ then*

$$\mathsf{blur}(\mathsf{partEval}(Pol, \Sigma, E), B) = \mathsf{blur}(\mathsf{partEval}(Pol, \Sigma', E), B)\,.$$

The precondition $E \cap B = \emptyset$ is very important; if it were violated, then some protected literal might be evaluated during filtering. If this happens, one can find counterexamples to the above theorem where some protected state literals can be deduced from the filtered policy.

Moreover, for a correct negotiation, $E \cup B$ *should cover all state literals that cannot be made true by the client.* This guarantees that the result of the filtering contains only predicates that can be understood and effectively handled by the client. This discussion gives us a method for determining B:

Let *LSL* be the set of all *local state literals*, that is, those with a predicate p such that

- p.`type` is `state_predicate`,
- p.`actor` is not `peer`

(a more formal definition is given in the next section.) Then let $B = LSL \setminus E$.

Note that both *LSL* and E are determined by the metadata, and hence B is, as well.

Another important question is: are there any pieces of *certain* information that the client may extract from a blurred program? More concretely:

- Can the client ever be sure that some credentials fulfill a request expressed as a blurred program? Then the client may prefer to send immediately such credentials, in order to minimize useless disclosure.
- Can the client detect when its credentials do not suffice to satisfy the server's request? Then the client may immediately abort the transaction, without any further unnecessary disclosure.

Fortunately, the answer to such questions in many cases is *yes*, and the reasoning needed to carry out this kind of analysis has the same complexity as plain credential selection, because reasoning boils down to computing two canonical models.

Theorem 3 *For all blurred policies Bpol, let $Bpol^{\max} = Bpol \cup \{\texttt{blurred}\}$ and $Bpol^{\min} = Bpol$. Then, for all states Σ and all sets of state predicates B,*

$$\begin{aligned}
&\mathsf{cmodel}(Bpol^{\max}, \Sigma) = \\
&\qquad \bigcup\{\mathsf{cmodel}(P, \Sigma) \mid \mathsf{blur}(P, B) = Bpol\}\,, \\
&\mathsf{cmodel}(Bpol^{\min}, \Sigma) = \\
&\qquad \bigcap\{\mathsf{cmodel}(P, \Sigma) \mid \mathsf{blur}(P, B) = Bpol\}\,.
\end{aligned}$$

Informally speaking, this theorem says that *Bpol* contains *all* the information that does not depend on blurred conditions. More precisely, the policies P such that $\mathsf{blur}(P, B) = \mathit{Bpol}$ are those that might have originated *Bpol*; $\mathit{Bpol}^{\min}$ captures the consequences that are true in all these possible policies P, and the complement of $\mathit{Bpol}^{\max}$ contains the facts that are false in all possible P.

As a corollary of the above theorem, every consequence of $\mathit{Bpol}^{\min}$ is also a consequence of the original non-blurred policy, and every atom that cannot be derived with $\mathit{Bpol}^{\max}$, cannot be derived from the non-blurred policy either. This is what the client can deduce from *Bpol*.

Blurring is used also to deal with delayed actions. Delayed provisional predicates must be evaluated after the response of the client, and in general cannot be understood by the client, just like private predicates. Therefore it is appropriate to treat delayed state predicates like private predicates. Nonetheless, distinguishing the two classes of predicates is useful to keep track of why their evaluation is delayed.

4.2 Pre-evaluating External Provisional Predicates

Let Pol_x be the set of *Pol*'s rules containing some external provisional literal. By Lemma **??**, we have

$$\begin{aligned}
&\mathsf{cmodel}(\mathit{Pol}, \Sigma) = \\
&\qquad = \mathsf{cmodel}(\mathit{Pol} \cup \mathit{Pol}_x, \Sigma) \\
&\qquad = \mathsf{cmodel}(\mathit{Pol} \cup \mathsf{partEval}(\mathit{Pol}_x, \Sigma, E), \Sigma)\,.
\end{aligned}$$

4.3 Action Gathering

5 Protune Filtering Process

On each party, the policy filtering process is determined by several parameters:

- a request *Req* from the client, requiring a decision about access control, or portfolio information release,
- an access control or portfolio release policy *Pol*,
- a metapolicy *Mpol*,
- the current state Σ.

With the exception of *Req*, all the parameters are local to the peer which is to make the decision. The metapolicy is evaluated against the current state, yielding the *current canonical metamodel MM*:

$$\mathit{MM} = \mathsf{cmodel}(\mathit{Mpol}, \Sigma)$$

which is inspected to read the metaproperties of rules and predicates.

As a result of the filtering process, the following two sets are returned:

- $FP \equiv$ a set of filtered policies
- $DA \equiv$ set of actions that are reffered and must be executed once the negotiation is finished

5.1 Filtering Process

$$S = \begin{cases} \emptyset & \text{if } \mathtt{L_i} \in \mathtt{X^{fail}}\ , \\ \{\mathtt{A} \leftarrow \mathtt{L_1}, \ldots, \mathtt{L_{i-1}}, \mathtt{Val}, \mathtt{L_{i+1}}, \ldots, \mathtt{L_n}\} & \text{if } \mathtt{L_i} \notin \mathtt{X^{fail}}. \end{cases}$$