Security-by-Contract for Web Services

or How to Trade Credentials for Services*

Nicola Dragoni Fabio Massacci
Department of Information and Communication Technologies
University of Trento, Italy
surname@dit.unitn.it

ABSTRACT

The classical approach to access control of Web Services is to present a number of credentials for the access to a service and possibly negotiate their disclosure using a suitable negotiation protocol and a policy to protect them.

In practice a "Web Service" is not really a single service but rather a set of services that can be accessed only through a suitable conversation. Further, in real-life we are often willing to trade the disclosure of personal attributes (frequent flyer number, car plate or AAA membership etc.) in change of additional services and only in a particular order.

In this paper we propose a novel negotiation framework where services, needed credentials, and behavioral constraints on the disclosure of privileges are bundled together and that clients and servers have a hierarchy of preferences among the different bundles.

While the protocol supports arbitrary negotiation strategies we sketch two concrete strategies (one for the client and one for the service provider) that make it possible to successfully complete a negotiation when dealing with a cooperative partner and to resist attacks by malicious agent to "vacuum-clean" the preference policy of the honest participant.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services, Commercial services*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms, Security, Theory

*Research partly supported by the project EU-IST-STREP-S3MS.

SWS'07, November 2, 2007, Fairfax, Virginia, USA.

Keywords

Security-by-Contract, Automated Trust Negotiation, Web Services

1. INTRODUCTION

The basic tenet of Service Oriented Computing is the possibility of building distributed applications on the Web by using Web Services (WS) as basic building blocks. If the set of related functionalities is significantly large and can only be accessed according a suitable work-flow then we more appropriately speak of WS Conversations or Business Processes for WS.

Controlling access to such services has become a key issue because services are distributed, and might also be controlled by different entities. Indeed, one often speak of *policy-based access control* of services (e.g. [18, 6, 13] or the large number of papers appeared in the IEEE Policy Workshop series and the ACM SACMAT series). The intuition is that access to services and resources is automatically derived from policies that are deployed on the point of service. Policies might specify the various attributes (often called privileges) that the individual clients might need to show in order to access to each service.

In order to identify the holders of the appropriate set of privileges across the web cryptographic credentials can be put in place thus creating what is called a Privilege Management Infrastructure or PMI [5] or, with a slightly misleading terminology, a trust management system [3, 19]. Credentials needed to access the system might be presented in push or pull mode or discovered interactively [11].

One of the key issue, as pointed out initially by Yu, Winslett and Seamons [20], is the problem of gradually building trust between two unknown parties so that the client doesn't empty its wallet on the counter and the server doesn't list an encyclopedia of policies rules (sometimes highly sensitive, most likely highly irrelevant).

Yet, Yu, Winslett, and Seamons [20], and many other works [16, 4, 12, 11] where the need of credentials is gradually disclosed to the client, take for granted that *a client always starts the negotiation by requesting access to* a *resource.*

Instead, as pointed out by Mecella et al. [14]:

> While many in the literature treated Web services as a set of independent single operations, interacting with real world Web services involves generally a sequence of invocations of several of their operations, referred to as conversation. A

simple example is a bookstore Web service; buying a book involves generally searching for the book, browsing the details and reviews about this book, adding the book to the shopping cart, checking out, paying, etc.

It is therefore important to consider the access control and negotiation issues for the overall WS conversation. As noted by Koshutanski and Massacci [10] it might well be that the conversation takes different routes, therefore changing the set of needed credentials. Keeping up with the book example, if we decided to send the books as gift then we only need to specify the address of the credit card holder and the address of the gift recipient. Our address is not needed.

Mecella et al. [14] have provided an access control model and a trust-negotiation scheme for WS where such conversational aspect is taken care of. While they take full care of the behavioral aspect of WS, their negotiation protocol still sticks to the progressive disclosure of credentials while keeping the set of requested services fixed.

What is missing is a typical feature of real-life negotiations: *we are usually willing to trade off disclosure of our security attributes for (additional) services.* Back to the book shopping example: we might not be willing to disclose our Frequent Flyer card for buying a book. But we might be willing to do so if the system tells us that travel books gets a 10% discount if a Frequent Flyer card is disclosed.

1.1 The Contribution of this Paper

In this paper we propose a negotiation framework that considers not only the negotiation of credentials but also the corresponding negotiation of services and the behavioral constraints on the disclosures of credentials depending on the business process.

In a nutshell we envisage that services and privileges are *bundled* together and that clients and servers have a *hierarchy of preferences* among the different bundles.

From the perspective of a client, say Alice, we assume that she has a ranked list of services and for each set of services in this rank she has a ranked list of security attributes or digital credentials that she is willing to disclose. For each set of attributes, she also has a behavioral model that dictates how she will disclose her credentials as soon as she uses the services. For example Alice might be willing to disclose her credit card in order to buy a book and her frequent flier number in order to have a discount. She might be willing to disclose her car plate number in order to buy technical manuals from the car manufacturer only available to holder of a specific brand of cars. A similar tree-like structure might be present at Bob's server.

Notice that we do not assume that the ranked list is complete (i.e. in the lists there are all possible sets) because it will not be realistic.

While negotiation protocols in the agent community [15] can assume cooperative agents, such assumption is not acceptable in a security setting. So we have designed the framework to take into account both cooperative and malicious agents.

In the next Section we introduce the overall framework which also defines the type of messages participants can exchange, the type of information messages will contain and finally we specify the general protocol flow (§3). In order to show that the flow is actually executable we define two possible strategies for the client and the service provider (§4). Such strategies are at the same time cooperative (they do negotiate a mutually liked bundle of services and privileges if one exists) and robust (they resist to potentially malicious parties that would only like to discover the preferences of their opponents). We conclude the paper with a discussion on related works and highlighting the main contributions of our proposal (§5).

2. THE NEGOTIATION FRAMEWORK

The basic idea behind our proposal is a combination of the idea of *programming-by-contract* originally introduced by B. Meyer for object oriented software and later applied to WS [9] and extended to *security-by-contract* as proposed by Dragoni et al. [7]. Similar ideas are also present in the definition of the WS security behavior by Mecella et al. [14].

In a nutshell a web service provider is offering a *contract*: I'll grant you the services $s_1, \ldots s_n$, but in change I want you to show me that you have security attributes (or privileges) $p_1, \ldots p_n$. Further, I'll ask you to show me your credentials according the following dynamic behavior ω_1 where e.g. possession of privilege p_i is asked before service s_j can be accessed.

On the other side a client is making a counter-proposal: I want to use your services $s'_1, \ldots s'_n$, and I'm only willing to give you evidence that I have security privileges $p'_1, \ldots p'_m$. Further, I'm going to accept showing my credentials only according the following dynamic behavior ω'_1 where e.g. I'm willing to show possession of privilege p_i but only is I'm asking service s_1 or s_2.

In our setting security attributes will usually be digitally signed credentials in X509 format [5] and requests for credentials can be provided by SAML tokens as proposed by Kohustanski and Massacci [11], but it is not necessarily the case. For example one can speculate that certain services are only available to mobile users and one resorts to techniques belonging to Mobile IPv6 security [2]. On the other side services could be described by WSLD or semantic web services in OWL.

For the formal theory that we develop in the rest of the paper, we assume them to be atomic predicates as in Yu, Winslett, and Seamons [20] and Koshutanski and Massacci [11]. Further, instead of referring to them as "security attributes or digital credentials to be disclosed" we will simply refer to them as *privileges* (from PMI).

DEFINITION 2.1. *Let $\mathcal{P}$ be a set of atomic proposition p denoting security privileges and let $\mathcal{S}$ be a set of atomic propositions s (disjoint from $\mathcal{P}$) denoting services.*

If contract represents the security behavior of a WS the temptation would be to make such contractual claims arbitrarily complex. Since we argue that contract should be matched and negotiated by the WS on-the-fly a complex procedure is likely to defy the spirit of our proposal. So we suggest to follow the ideas behind a number of papers [17, 14, 7] and represent the security behavior as an automaton.

DEFINITION 2.2. *The set of security behaviors Ω is a finite state automaton whose actions are drawn from $\mathcal{P}$ and $\mathcal{S}$.*

From now on, let us refer to a service client as Executor and to a service provider as Provider. We have iden-

tified the following abstract operator (Ω^P and Ω^E indicate respectively the behavior of Provider and Executor):

- [Traces Operator] $\mathcal{T} = \mathsf{Traces}(\Omega)$
  It returns the set $\mathcal{T}$ of all the possible sequences of actions that can be performed according to the security behavior Ω.

- [Match Operator $\sqsubseteq$] $\Omega^E \sqsubseteq \Omega^P$
  It returns 1 if the behavior specified by Ω^E is among the behaviors that are allowed by Ω^P, 0 otherwise.
  In terms of the Traces operator:
  $\Omega^E \sqsubseteq \Omega^P \Leftrightarrow \mathsf{Traces}(\Omega^E) \subseteq \mathsf{Traces}(\Omega^P)$

From now on we will say that *the security behavior Ω^E of Executor matches the security behavior Ω^P of Provider* if and only if $\Omega^E \sqsubseteq \Omega^P$ returns *true*. A detailed discussion on such operator and its possible implementation is outside the scope of the paper. However, interested readers can find in [7] detailed algorithms for matching security behaviors.

We assume agents have preferences over different negotiation alternatives, or proposals.

DEFINITION 2.3. *Let $\langle S^A, P^A, \Omega^A \rangle$ be a tuple representing a* proposal of a generic agent *A, where $S^A \subset \mathcal{S}$ is a set of services, $P^A \subset \mathcal{P}$ a set of privileges and Ω^A a security behavior.*

DEFINITION 2.4. *Let PS^A be the set of tuples $\langle S^A, P^A, \Omega^A \rangle$ representing* the policy space *of A.*

Each Executor E , resp. Provider P , will have his own policy spaces PS^E, resp. PS^P. We also assume that preferences are specified by means of a partial order $\ll_E$ over PS^E (resp. $\ll_P$ over PS^P).

According to the above definitions, from now on we will use $\langle S^E, P^E, \Omega^E \rangle$ and $\langle S^P, P^P, \Omega^P \rangle$ to indicate a proposal of Executor and Provider, respectively.

DEFINITION 2.5. *$\langle S, P, \Omega \rangle$ is an* acceptable proposal for an Executor E *if there exists an $\langle S^E, P^E, \Omega^E \rangle$ in PS^E such that $S \supseteq S^E$, $P \subseteq P^E$, and $\Omega \sqsubseteq \Omega^E$. $\langle S^E, P^E, \Omega^E \rangle$ will be a* solution for the Executor E .

DEFINITION 2.6. *$\langle S, P, \Omega \rangle$ is an* acceptable proposal for a Provider P *if there exists a $\langle S^P, P^P, \Omega^P \rangle$ in PS^P such that $S^P \supseteq S$, $P^P \subseteq P$, and $\Omega^P \sqsubseteq \Omega$. $\langle S^P, P^P, \Omega^P \rangle$ will be a* solution for the Provider P .

Notice that an acceptable proposal for a provider is not necessary an acceptable proposal for an executor.

DEFINITION 2.7. *$\langle S, P, \Omega \rangle$ is the* best solution *for the agent A if:*

1. *$\langle S, P, \Omega \rangle$ is a solution for A and*
2. *there is no another solution $\langle S', P', \Omega' \rangle$ such that $\langle S', P', \Omega' \rangle \ll_A \langle S, P, \Omega \rangle$.*

Table 1 shows the type of messages agents can exchange during the overall negotiation protocol as well as their meaning. Essentially, agents send or accept a proposal by means of the propose and accept message, respectively. The ask message is used by an Executor to start a negotiation and the failure and no_more_proposals messages are used for terminating the negotiation when a protocol violation occurs or an agent cannot proceed for some reason (see Section 3 for details).

Table 1: Messages and their Meaning

Message	Meaning
ask(S^E)	Executor asks Provider for services S^E
propose(S^E, P^E, Ω^E)	Executor wants at least services S^E, gives at most privileges P^P and accepts at most to behave as Ω^E.
propose(S^P, P^P, Ω^P)	Provider offers services S^P, requires at least privileges P^P and promises at most to behave as Ω^P.
accept(S^P, P^E, Ω^E)	Agent (Provider or Executor) accepts services S^P, privileges P^E, and security behavior Ω^E.
no_more_proposals(S^E)	Agent (Provider or Executor) has no more proposals for the negotiation of S^E.
failure	A protocol violation has occurred or the negotiation terminates unsuccessfully.

To guarantee safety and timely termination of trust negotiation no matter what policies the parties possess, our protocol requires the negotiation strategies used with it to enforce the following conditions throughout negotiations.

Proposal Conditions.

Let $\mathbb{P}^E_{sent}$, $\mathbb{P}^P_{sent}$ be the set of proposals sent by Executor and Provider at a given point in the negotiation process, respectively. To send a propose(S^E, P^E, Ω^E) (resp., propose(S^P, P^P, Ω^P)) message, the following conditions must hold:

1. $S^P \supseteq S^E$
   From the point of view of Provider, this means that each proposal must contain at least the services required by Provider. For Executor, this means that it can not *propose* different services. To do this, Executor must exploit the ask message, restarting in this way the negotiation on a new set of services.
2. $(P^E \neq \emptyset) \wedge (\Omega^E \neq \emptyset)$ $\quad$ $((P^P \neq \emptyset) \wedge (\Omega^P \neq \emptyset))$
   To avoid attacks that could cause the complete disclosure of a policy, both privileges and acceptable security behaviors must not be empty.
3. $\langle S^E, P^E, \Omega^E \rangle \notin \mathbb{P}^E_{sent}$ $\quad$ $(\langle S^P, P^P, \Omega^P \rangle \notin \mathbb{P}^P_{sent})$
   A proposal can be sent at most once. Again, this condition prevents the full disclosure of an agent's policy caused by possible attacks.

Remark. The above conditions guarantee attack resistance by requiring some *progress* in the negotiation process each time a proposal is sent by a negotiator. In other words, an agent gradually discloses its policy if and only if the other agent does the same, sending new proposals at each negotiation step (that is, each time a propose message is received).

Acceptance Conditions.

Let $\mathbb{P}^E_{received}$, $\mathbb{P}^P_{received}$ be the set of proposals received by Executor and Provider at a given point in the negotiation process, respectively. To send an accept(S^P, P^E, Ω^E) message, the following conditions must hold:

- Executor: there exists $\langle S^P, P^P, \Omega^P \rangle$ such that
  1. $\langle S^P, P^P, \Omega^P \rangle \in \mathbb{P}^E_{received} \wedge$
  2. $\langle S^P, P^P, \Omega^P \rangle$ is an acceptable proposal with solution $\langle S^E, P^E, \Omega^E \rangle$
- Provider: there exists $\langle S^E, P^E, \Omega^E \rangle$ such that
  1. $\langle S^E, P^E, \Omega^E \rangle \in \mathbb{P}^P_{received} \wedge$
  2. $\langle S^E, P^E, \Omega^E \rangle$ is an acceptable proposal with solution $\langle S^P, P^P, \Omega^P \rangle$

In other words, the above conditions require that both Executor and Provider can accept only a received acceptable proposal. Note that we do not require that the acceptable proposal is the last received proposal, but only that it was received in the past. Indeed, we cannot force an agent to accept the first acceptable proposal he receives because this would prevent the agent to negotiate for a better solution.

3. PROTOCOL FLOW

The intuition behind the protocol is that agents continuously exchange propose messages until an accept or failure message is sent by one party (terminating successfully or unsuccessfully the protocol). For the sake of readability and due to space limitations, we split the protocol flow in several Figures (1, 2, 3 and 4).

The protocol works as follows:

- Executor starts the protocol sending the Provider a request of services S^E by means of an ask(S^E) message (Figure 1).
- Provider can reply with one of the following two messages:
  - no_more_proposals(S^E): Provider has no more proposals $\langle S^P, P^P, \Omega^P \rangle$ for S^E. In this particular case, this means that Provider does not have any bundle S^P which contains S^E.
  - propose(S^P, P^P, Ω^P): Provider offers the bundle S^P (containing S^E), it requires at least privileges P^P and it promises to behave at most as described by Ω^P.
- When Executor receives a propose message, it can reply with one of the following messages:
  - propose(S^E, P^E, Ω^E): Executor sends a new proposal for services S^E. Note that this new proposal must follow the *proposal conditions*.
  - accept(S^P, P^E, Ω^E): Executor accepts a Provider's proposal and the protocol ends successfully. Note that the *acceptance conditions* must hold, so we are sure that Executor is accepting an acceptable proposal previously sent by Provider. We stress that the acceptable proposal could not be the last one made by Provider, since Executor could decide to postpone an accept message until it has no more counterproposals to send.
  - failure: this message is sent if some protocol violation occurs, that is the received propose message does not satisfy the *proposal conditions*. The protocol ends unsuccessfully.
  - no_more_proposals(S^E): the Executor has no more proposals for S^E.
- When Executor receives a no_more_proposals(S^E) message (Figure 2), it can reply with:
  - accept(S^P, P^E, Ω^E): Executor accepts an acceptable proposal previously received. Again, the *acceptation conditions* must hold and the protocols ends successfully.
  - ask(S^E_{new}): Executor asks for a new set of services S^S_{new}, restarting the protocol.
  - failure: a failure message is sent if Executor has no more services to ask for. The protocol ends unsuccessfully.
- When Provider receives a propose message the possible answers are the same as those of Executor, as shown in Figure 3.
- Finally, a Provider can reply to a no_more_proposals(S^E) message (Figure 4) with:
  - accept(S^P, P^E, Ω^E): subjected to the acceptation conditions already stated. The protocol ends successfully.
  - failure: if Provider has no solution for S^E. In particular, this means that Provider didn't receive any acceptable proposal from Executor. The protocol ends unsuccessfully.

Protocol Termination..

The negotiation ends successfully when an accept message is sent by an agent. Instead, the protocol ends unsuccessfully when an agent sends a failure message.

4. NEGOTIATION STRATEGIES

A *Services-vs-Privileges Negotiation Strategy* controls the exact content of messages, i.e. which proposals of the form $\langle S, P, \Omega \rangle$ agent disclose and when disclose them. For the sake of readability and simplicity, in this Section we describe only the main ideas behind the Executor and Provider strategies, omitting technical and implementation details. The programs implement the protocol flow as well as the agents' strategies we are going to discuss.

Let j, k, q be indexes referring to some sets S, P, Ω, respectively. To describe an agent's strategy, we assume agent's policy is structured as shown in Figure 5. Basically, a policy is represented by a $SP\Omega$-**structure** composed by several $SP\Omega$-**trees**, that is trees having a bundle S_j as root, then a level of privileges set P_{jk} and finally a level of security rule sets Ω_{jkq}. The $SP\Omega$-structure contains the policy space of an agent, say A, and it is ordered according to the relation $\ll_A$.

An important remark is that, to describe the strategy of one party (Executor or Provider), we assume that *its* own policy is represented as a $SP\Omega$-structure, but we do not necessarily require the same for the other party. This because

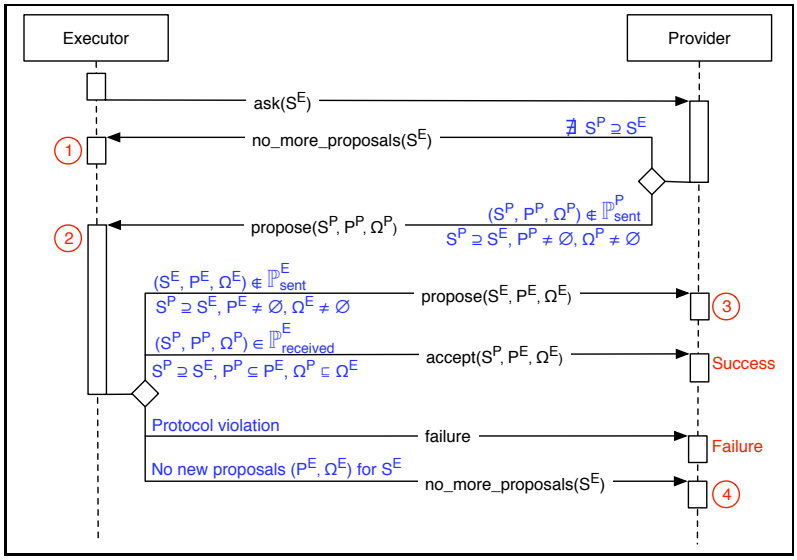


Figure 1: Trust Negotiation Protocol (part I)

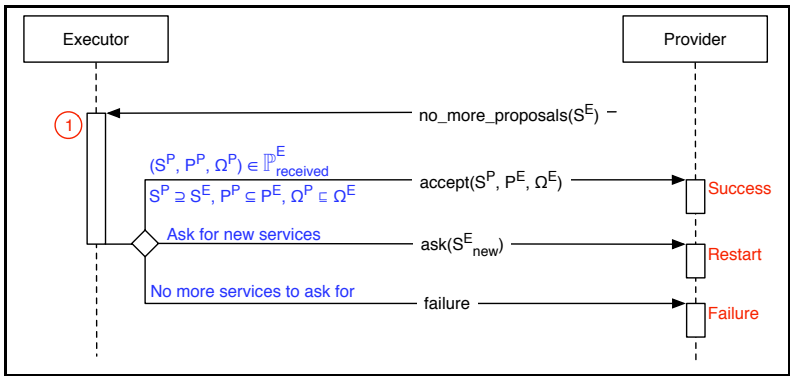


Figure 2: Executor: possible replies to a no_more_proposals message.

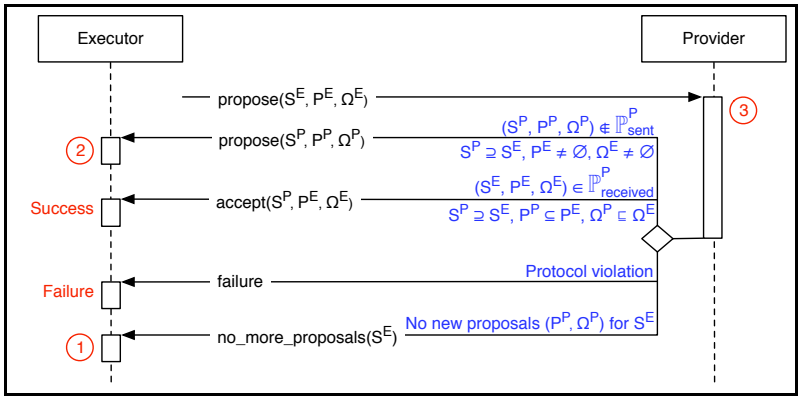


Figure 3: Provider: possible replies to a propose message.

we consider the other party as a *self interested agent* which most reasonably will follow its own (best) strategy[1]. In particular, we assume that one party does not know which policy's representation and reasoning (i.e., search strategy in the policy search space) the other party will follow. The only information an agent knows of the other party is the

[1]Our protocol does not impose any strategy to agents.

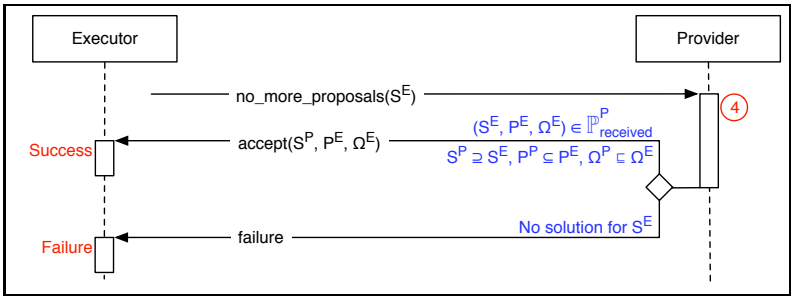


Figure 4: Provider: possible replies to a no_more_proposals message.

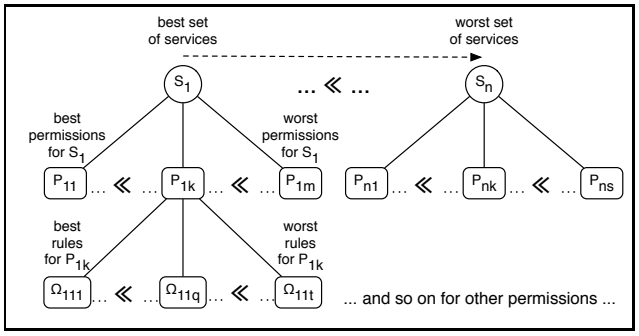


Figure 5: Agent's Policy as $SP\Omega$-structure.

set of messages it receives. The foregoing assumption has a significant impact on the design of an agent strategy, because knowing more information could improve the strategy by reducing the number of interactions.

Of course, given a negotiation protocol agents must follow, the number of possible strategies is almost limitless and the definition of a specific strategy usually depends on which properties the strategy wants to guarantee. In this paper, we have designed the agents' strategies with two properties in mind.

PROPERTY 4.1. *(Sound Termination) Assuming cooperative and rational agents, if a solution exists then eventually agents will reach an agreement and the protocol will end successfully.*

In other words, if agent are cooperative and rational, then they will agree on a mutually liked bundle of services and privileges if one exists.

The second property we want to guarantee is that, assuming self interested but not necessarily cooperative agents, strategies must be robust, that is they must resist to potentially malicious parties that would only like to discover the preferences of their opponents:

PROPERTY 4.2. *(**Attack Resistance**) Agents does not disclose their preferences to malicious parties.*

4.1 Strategies

According to the protocol flow described in Section 3, Executor starts the negotiation sending Provider a request of services S_j^E and then it waits for Provider's reply. It is reasonable to assume that Executor will start asking for his best bundle of services, i.e., S_1^E.

Received a request for services S^E, Provider searches in his $SP\Omega$-structure for the best bundle S^P such that $S^P \supseteq S^E$. If this is available, it replies with his best proposal, i.e. $\langle S_j^P, P_{j1}^P, \Omega_{j11}^P \rangle$, otherwise it sends a no_more_proposals(S^E) message.

From this point on, agents negotiate by exchanging propose messages that satisfy the proposal conditions. According to the protocol flow, this exchange ends when one party accepts a proposal of the other party or some failure occurs. The strategies followed by both agents when they receive a propose message is summarized in the following EVALUATE_PROPOSAL procedure, where parameters S, P, Ωrepresent the received proposal and j is the index of the current bundle of services S_j:

```
1: procedure EVALUATE_PROPOSAL(S, P, Ω, j)
2:     Find a counterproposal for S_j
3:     if counterproposal does not exist then
4:         Find best proposal for S_j
5:     Find a solution for S_j
6:     Store solution if better than the previous one
7:     if both proposal and solution exist then
8:         if proposal ≪ solution then
9:             sendMsg(propose(proposal))
10:        else
11:            sendMsg(accept(solution)) // Success
12:    else if proposal exists then
13:        sendMsg(propose(proposal))
14:    else if solution exists then
15:        sendMsg(accept(solution)) // Success
16:    else
17:        sendMsg(no_more_proposals(S_j))
```

The basic idea behind the above strategy is that, given a proposal sent by the other party, an agent searches in his $SP\Omega$-Structure for both a new proposal and a solution. If a solution is found, then it is stored if better than the old one. If both proposal and solution exist, then the agent will act following the best behavior according to his $SP\Omega$-structure, that is it will send a new proposal if better than the solution (proposal $\ll$ solution in the $SP\Omega$-structure) or it will accept the solution otherwise (solution $\ll$ proposal in the $SP\Omega$-structure). In this way we are sure that an agent will always act trying to get his best, but also that if a solution exists than sooner or later it will be proposed by a party and accepted by the other (as described in the protocol flow in Section 3).

4.1.1 Selection of a Proposal

According to the above strategy, an agent first searches for

his best (not already proposed) *counterproposal* for S_j, that is a proposal such that $P^E \subset P^P$. From the point of view of Executor, this means reducing the privileges requested by Provider, while for Provider this means asking for more privileges respect to those given by Executor. A sketch of the FIND_COUNTERPROPOSAL function of Provider follows, where FIND_BEST_CONTRACT is a function that returns the best (not already proposed) contract under a given privilege set.

```
1: function FIND_COUNTERPROPOSAL(P^E, j, k)
2:     Search for P^P_jk such that P^E ⊂ P^P_jk
3:     if P^P_jk ≠ ∅ then
4:         Ω^P_jkq ⇐ FIND_BEST_CONTRACT(j, k, 1)
5:         if Ω^P_jkq ≠ ∅ then
6:             return(P^P_jk, Ω^P_jkq)
7:         else
8:             return(FIND_COUNTERPROPOSAL(P^E, j, k+1))
9:     else
10:        return(∅, ∅)
```

An example of counterproposal is shown in Figure 6. The algorithm visits the privileges sets in a BFS way from the left to the right (i.e., from the best to the worst privilege set) looking for the first P^P_{jk} such that $P^E \subset P^P_{jk}$. Then it goes down to one level searching for the best (not already proposed) contract Ω. If all the contracts have been already proposed, a new privilege set is searched starting from P^P_{jk+1}. This is the case of P_{11} in the Figure, since it is a counterproposal ($P_{11} \supset P^E$), but it is not selected because all the contracts under it have been already sent (i.e., proposal $\langle S_1, P_{11}, \_\rangle$ cannot be sent). Therefore, the best counterproposal is $\langle S_1, P_{13}, \Omega_{132}\rangle$ (note also that Ω_{131} can not be chosen because already proposed).

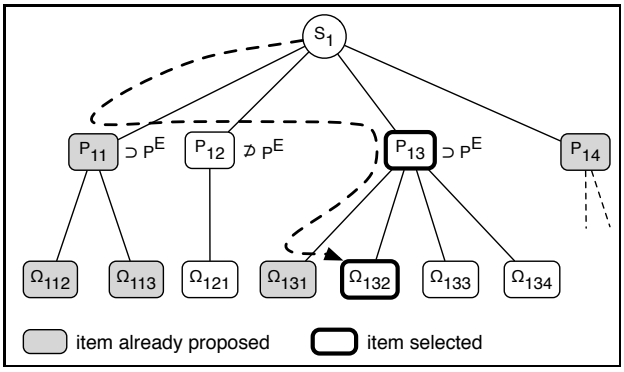


Figure 6: Example of Counterproposal

If a counterproposal is not available, then the agent searches for his best (not already sent) proposal for S^E_j. An example of best proposal is shown in Figure 7, where the policy is the same as the one of the previous example (note that the best proposal is different from the counterproposal).

A sketch of the FIND_BEST_PROPOSAL function of Provider follows.

```
1: function FIND_BEST_PROPOSAL(j, k)
2:     if P^P_jk ≠ ∅ then
3:         Ω^P_jkq ⇐ FIND_BEST_CONTRACT(j, k, 1)
4:         if Ω^P_jkq ≠ ∅ then
5:             return(P^P_jk, Ω^P_jkq)
6:         else
7:             return(FIND_BEST_PROPOSAL(j, k+1))
8:     else
9:         return(∅, ∅)
```

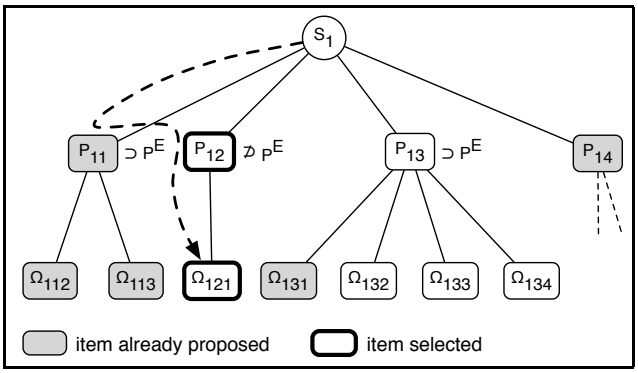
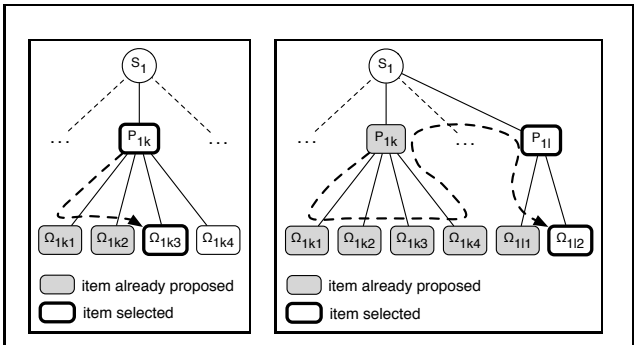


Figure 7: Example of Best Proposal

In both cases (counterproposal and best proposal), security rules are determined by searching for the best (not already proposed) security rule set Ω, as shown in Figure 8. This is done by using the FIND_BEST_CONTRACT function cited above. If all security rule sets Ωunder a privilege set Phave been already proposed, a new privilege set $P$is searched (according to the current strategy, i.e. counterproposal or best proposal).



(a) Ωavailable under the current P

(b) Ωavailable under a different P

Figure 8: Selection of the Best Security Rule Set

Remark. At this stage we assume that agents negotiate in a *cooperative* way, i.e., proposing different privileges only if a counterproposal is not available.

4.1.2 Selection of a Solution

The strategies exploited by Provider and Executor for finding a solution are basically the same, except for the

search space they visit. Indeed, Executor just needs to search the solution in the $SP\Omega$-tree corresponding to the requested set of services S^E, while Provider could need to search the solution in different $SP\Omega$-trees having as root a bundle $S^P \supseteq S^E$. The Provider must search for a solution in another $SP\Omega$-tree if it has not found any solution in the current $SP\Omega$-tree. Functions FIND_SOLUTION_IN_TREE and FIND_SOLUTION implement the strategy for Provider while Executor uses only the function FIND_SOLUTION_IN_TREE.

1: **function** FIND_SOLUTION_IN_TREE(P^E, Ω^E, j, k)
2: Search for $P^P_{jk} : P^P_{jk} \subseteq P^E$
3: **if** $P^P_{jk} \neq \emptyset$ **then**
4: $\Omega^P_{jkq} \Leftarrow$ FIND_CONTRACT(Ω^E, j, k, 1)
5: **if** $\Omega^P_{jkq} \neq \emptyset$ **then**
6: return(P^P_{jk}, Ω^P_{jkq})
7: **else**
8: return(FIND_SOLUTION_IN_TREE(P^E,Ω^E, j, k+1))
9: **else**
10: return($\emptyset$, $\emptyset$)

11: **function** FIND_SOLUTION(S^E, P^E, Ω^E, j, k)
12: (P^P_{jk}, Ω^P_{jkq}) $\Leftarrow$ FIND_SOLUTION_IN_TREE(P^E, Ω^E, j, k)
13: **if** $P^P_{jk} = \emptyset$ **then**
14: $S^P_j \Leftarrow$ FIND_BEST_SERVICE(S^E, j)
15: **if** $S^P_j \neq \emptyset$ **then**
16: $P^P_{jk} \Leftarrow$ FIND_SOLUTION(S^E, P^E, Ω^E, j, 1)
17: return(S^P_j, P^P_{jk}, Ω^P_{jkq})

Starting from an index j in the $SP\Omega$-structure, the function FIND_BEST_SERVICE searches for and returns the best set S^P_t containing S^E, with t $\geq$ j. The function FIND_CONTRACT searches for the best contract Ω^P_{jkq} (under a privilege set P^P_{jk}) such that $\Omega^P_{jkq} \sqsubseteq \Omega^E$.

The search strategy of FIND_SOLUTION_IN_TREE is the same as the one exploited to find a counterproposal: the first $P^P_{jk} : P^P_{jk} \subseteq P^E$ is searched in a BNF way and then the algorithm calls FIND_CONTRACT to search for the best contract Ω^P_{jkq} not already proposed such that $\Omega^P_{jkq} \sqsubseteq \Omega^E$. If the contract is not found under P^P_{jk}, then the search restarts from P^P_{jk+1}.

EXAMPLE 4.1. *In the negotiation of Figure 9, a solution exists but Provider does not accept it until it receives a* no_more_proposals *message from Executor. Indeed, Provider always sent a proposal because better than that solution.*

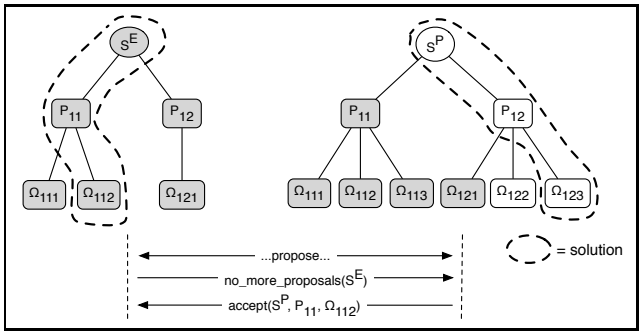


Figure 9: Example 4.1

EXAMPLE 4.2. *In this Example, Provider receives an acceptable proposal from Executor and accepts it because the solution it finds is better than any possible remaining proposals.*

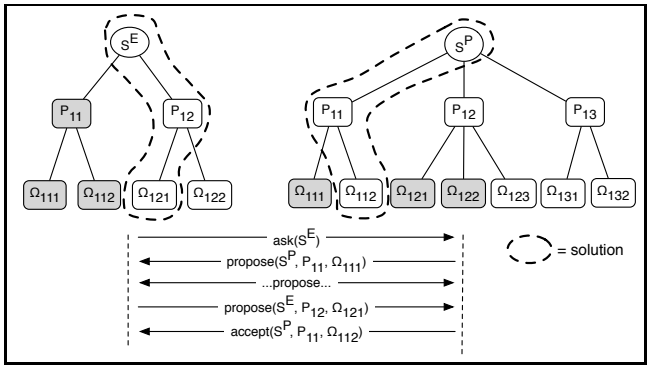


Figure 10: Example 4.2

4.1.3 Ensuring Strategy's Properties

The foregoing strategy ensures the *Sound Termination* property by searching a solution each time the agent receives a proposal. If the solution is found then it is stored if better than the old one. In this way we are sure that, if a solution exists, then sooner or later the agent will accept it. Note that this could happen at the latest when the agent receives a no_more_proposals message from the other party (as described by the protocol flow in Section 3). Indeed, before the receipt of this message, the agent will always send a proposal if better than the current solution, trying to get his best.

The *Attack Resistance* property is ensured by checking the proposal conditions each time the agent receives a propose message. We stress that those conditions guarantee attack resistance by requiring some *progress* in the negotiation process. Therefore, the agent gradually discloses its policy if and only if the other agent does the same, sending new proposals at each negotiation step. As an example, the following CHECK_PROPOSAL function is used by Provider to perform this check, where $\langle S^E, P^E, \Omega^E \rangle$ represents the received proposal and S^P_j is the current bundle of services of Provider. Provider will continue the negotiation if and only if this function will return *true*.

1: **function** CHECK_PROPOSAL(S^E, P^E, Ω^E, S^P_j)
2: **if** $(S^P_j \supseteq S^E) \wedge (P^E \neq \emptyset) \wedge (\Omega^E \neq \emptyset) \wedge \langle S^E, P^E, \Omega^E \rangle \notin \mathbb{P}^P_{received}$ **then**
3: return(true)
4: **else**
5: return(false)

5. RELATED WORKS AND CONCLUSIONS

Regulating access control over distributed systems (such as the web) has been the subject of intense research in the last few years. A classical way is to set up Privilege Management Infrastructures such as PERMIS [5], SPKI [8] or other hybrid models [1]. Trust management systems are just a different name for such PMIs where more sophisticated rules for access are used [3, 19]. In such systems credentials needed to

access the system might be presented in push or pull mode or discovered interactively [11].

The controlled disclosure of such credentials can be the subject of a complex negotiation protocol [20, 16, 4, 12, 11]. However all those works, including more recent ones appearing in the POLICY workshop this year still consider as a starting point of the negotiation the request for access to a single service. Mecella et al. [14] pointed out the importance to consider the access control and negotiation issues for the overall business process taking into account the dynamic aspect of the conversation.

One of the limitation is essentially no paper considers the possibility of negotiating the disclosure of our privileges for (additional) services. This gap is also evident in the broad literature on service negotiations in multi-agent systems. Here the focus is mainly on one-to-many negotiation protocols for voting, auctions, bargaining and coalition formation (see [15] for a nice survey on the topic), therefore with emphasis on the negotiation strategy and how this strategy works when agents exploit some utility space (i.e., linear, nonlinear). Again, to the best of our knowledge, we don't know papers which address the aforementioned problem.

The contribution of this paper is twofold. First, we have proposed a negotiation framework that considers not only the negotiation of privileges but also the corresponding negotiation of services and the behavioral constraints on the disclosures of such privileges depending on the business process. In our framework services and required privileges are bundled together and clients and servers have a hierarchy of preferences among the different bundles. The framework also defines the type of messages participants can exchange during the overall negotiation, the type of information messages will contain and the general protocol flow.

As second contribution of the paper, we have designed two possible strategies for the client and the service provider in order to show that the flow is actually executable. To take into account both cooperative and malicious agents, we have designed such strategies to be at the same time cooperative (they do negotiate a mutually liked bundle of services and credentials if one exists) and robust (they resist to potentially malicious parties that would only like to discover the preferences of their opponents).

6. REFERENCES

[1] C. Altenschmidt, J. Biskup, U. Flegel, and Y. Karabulut. Secure mediation: Requirements, design, and architecture. *JCS*, 11(3):365–398, 2003.

[2] T. Aura and M. Roe. Designing the mobile ipv6 security protocol. *Annales des Télécommunications*, 61(3-4):332–356, 2006.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of IEEE SS&P'96*, pages 164–173. IEEE Press, 1996.

[4] P. Bonatti and P. Samarati. A unified framework for regulating access and information release on the web. *JCS*, 10(3):241–272, 2002.

[5] D. W. Chadwick and A. Otenko. The PERMIS X.509 role-based privilege management infrastructure. In *Proc. of SACMAT'02*, pages 135–140. ACM Press, 2002.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. of POLICY'01*, pages 18–38. Springer, 2001.

[7] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of EuroPKI'07*, pages 297–312. Springer-Verlag, 2007.

[8] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, September 1999. IETF RFC 2693.

[9] R. Heckel and M. Lohmann. Towards contract-based testing of web services. In *Proc. of the Int. Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 82 of *ENTCS*. Elsevier Sci., 2004.

[10] H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 15–24. ACM Press, 2003.

[11] H. Koshutanski and F. Massacci. A negotiation scheme for access rights establishment in autonomic communication. *J. of Net. and Sys. Managment*, 15(1):117–136, 2007.

[12] J. Li, N. Li, and W. H. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proc. of CCS'05*, pages 46–57. ACM Press, 2005.

[13] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy based framework for network services management. *J. of Net. and Sys. Managment*, 11(3):277–303, 2003.

[14] M. Mecella, M. Ouzzani, F. Paci, and E. Bertino. Access control enforcement for conversation-based web services. In *Proc. of WWW'06*, pages 257–266. ACM Press, 2006.

[15] T. Sandholm. Distributed rational decision making. In G. Weiss, editor, *Multiagent Systems*, pages 201–259. The MIT Press, Cambridge, Massachusetts, 1999.

[16] K. Seamons and W. Winsborough. Automated trust negotiation. Technical report, US Patent and Trademark Office, 2002. IBM Corporation, patent application filed March 7, 2000.

[17] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of ACM SOSP'03.*, pages 15–28. ACM Press, 2003.

[18] M. Sloman and E. Lupu. Policy specification for programmable networks. In *Proce. of the 1st Int. Working Conf. on Active Networks*, pages 73–84. Springer, 1999.

[19] W. Yao. *Trust management for widely distributed systems*. PhD thesis, Univ. of Cambridge, Computer Laboratory, 2004. Technical report UCAM-CL-TR-608, ISSN 1476-2986.

[20] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *TISSEC*, 6(1):1–42, 2003.