Filling the gap between Requirements Engineering and Public Key/Trust Management Infrastructures*

Paolo Giorgini[1], Fabio Massacci[1], John Mylopoulos[1,2], and Nicola Zannone[1]

[1] Department of Information and Communication Technology
University of Trento - Italy
{massacci,giorgini,zannone}@dit.unitn.it
[2] Department of Computer Science
University of Toronto - Canada
jm@cs.toronto.edu

Abstract. The last years have seen a major interest in designing and deploying trust management and public key infrastructures. Yet, it is still far from clear how one can pass from the organization and system requirements to the actual credentials and attribution of permissions in the PKI infrastructure.
Our goal in this paper is filling this gap. We propose a formal framework for modeling and analyzing security and trust requirements, that extends the Tropos methodology for early requirements modeling. The key intuition that underlies our work is the identification of distinct roles for actors that manipulate resources, accomplish goals or execute tasks, and actors that own or permit usage of resources or goals. The paper also presents a simple case study and a PKI/trust management implementation.

Keywords: Security Engineering, Modelling and Architecture, Verification, Privilege Management, PKI and eHealth applications, PKI Requirements Analysis, Trust.

1 Introduction

Trust Management and PKIs are hot topics in security research [5, 4, 7, 10, 14, 18]. There are a number of sophisticated policy languages (e.g., [8]), algorithms, and system for managing security credentials. The trust-management approach has a number of advantages over other mechanisms for specifying and controlling authorization, especially when security policy is distributed over a network or is otherwise decentralized.

Solutions based on public-key cryptography and credential have been shown to be well suited in satisfying the security requirements of distributed systems and becoming the foundation for those applications that require security authentication. The reason is that it is impractical and unrealistic to expect that each user in a large scale system has a previously established relationship with all other users.

* This work has been partially funded by the IST programme of the EU Commission, FET under the IST-2001-37004 WASP project and by the FIRB programme of MIUR under the RBNE0195K5 ASTRO Project. We would like to thank the anonymous reviewers for useful comments.

However, if we look at the connection between these credential-based systems and the requirements of the entire IT system we find a large gap. There are no methodologies for linking security policy to the mainstream requirements analysis process. This might be an instance of the general problem of security engineering. The usual approach towards the inclusion of security within a system is to identify security requirements after system design. This is a critical problem [2], mainly because security mechanisms have to be fitted into a pre-existing design which may not be able to accommodate them [25]. Late analysis of security requirements can also generate conflicts between security needs and functional requirements of the system. Even with the growing interest in secure engineering, current methodologies for information system development do not meet the needs for resolving the security related IS problem [26].

In the literature there are proposals improving on secure engineering (see [13, 15, 21, 23]) or architectures for trust management (see [5, 4, 14, 18, 22]), but nobody has proposed a methodology that considers together both these approaches. Our goal is to introduce a trust management system into the requirements engineering framework. Essentially, we would like to avoid designing an entire IT system and then retrofitting a PKI on its top, when it is already to late to make it fits snugly.

In this paper we introduce a process that integrates trust, security and system engineering, using the same concepts and notations used for requirements specification. To devise the PKI/trust management structure, we propose to proceed in three steps. First, we build a functional requirements model where we show functional dependencies among actors, then we give a trust requirements model, where we study whether trust relationships among actors correspond to security requirements. Finally, we built a PKI/trust management implementation where the designer can define credentials and delegations certificates confronting them with the relationships captured in the other models and checking whether an actor that offers a service is authorized to have it.

The paper is organized as follows. Next (§2) we provide an brief description of Tropos methodology and describe the basic concepts and diagrams that we use for modeling security. We introduce a simple Health Care Information System (§3) that will be used as case study throughout the paper. Then we present in the formalization of the security notions (§4) and define axioms and properties of our framework (§5). Next (§6) we introduce negative authorizations. Then (§7) we define the trust implementation of our framework into the RT framework. Finally, we conclude the paper with some directions for future work (§8).

2 Security-Aware Tropos

The first step towards security engineering is to model the entire organization and procedures. Security failures often are organizational or procedural failures [2]. Thus, we have chosen a requirement framework that allows for the modeling of the entire organization: Tropos [6].

Tropos is an agent-oriented software development methodology, tailored to describe both the organization and the system. One of the main feature of Tropos is the crucial role given to the early requirements analysis that precedes the prescriptive requirements specification. The main advantage of this is that, by doing an earlier analysis, one can

capture not only the *what* or the *how*, but also the *why* a piece of software is developed. This, in turn, supports a more refined analysis of the system dependencies and, in particular, for a much better and uniform treatment, not only of the system's functional requirements, but also of the non-functional requirements (the latter being usually very hard to deal with).

Tropos uses the concepts of actor, goal, soft goal, task, resource and social dependency for defining the obligations of actors (dependees) to other actors (dependers). Actors have strategic goals and intentions within the system or the organization and represent (social) agents (organizational, human or software), roles or positions (that represent a set of roles). A goal represents the strategic interests of an actor. A task specifies a particular course of action that produces a desired effect, and can be executed in order to satisfy a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource. In the rest of the paper, we say service for goal, task, or resource. For example, Yu et al. [20] have used the Tropos framework to model strategic goal concerning privacy and security of agents and have used formal tools for some reachability and goal analysis.

Although Tropos is based on the agent paradigm, it can be also combined with non-agent (e.g., object-oriented) software development paradigms. For example, one may want to use Tropos for early development phases and then use UML-based approaches (e.g., the Rational Unified Process). Tropos can be also combined with more formal approaches, like for instance [3], allowing so for the description of the dynamic aspects and the verification of requirements specification. There is a considerable amount of work in this direction within the Formal Tropos project [11].

After the Tropos formalization we are still for behind capturing security requirements, because Tropos has been designed with cooperative information systems in mind. Thus a dependency between two actors means that the dependee will take responsability for fulfilling the functional goal of a depender, but we have no way to specify or check that it is actually authorized to do so. Thus, we identify four relationships:

- *trust*, among two agents and a service,
- *delegation*, among two agents and a service,
- *ownership*, between an agent and a service, and
- *offer*, between an agent and a service.

Note the difference between owning a service and offering a service. For example, a patient is the legitimate owner of his personal data. However, the data may be stored on a medical information system that offers access to the data. This distinction explains clearly why IS managers need the consent of the patient for data processing. Also note the difference between trust and delegation. Delegation marks a formal passage in the requirements modeling. In contrast, trust marks simply a social relationship that is not formalized by a "contract" (such as digital credential).

Moreover, we do not assume that a delegation implies a trust. Using this extension of the modeling framework, we can now refine the process:

1. define functional dependencies of services among agents;
2. design a trust model among the actors of the systems;
3. identify who owns services and who is able to fulfill them.

3 An Illustrative Case Study

We present a simple case of health care IS to illustrate our approach. This example derives from EU privacy legislation: a citizen's personal data is processed by an information system (which offer a data access service) but it is owned by the citizen himself whose consent is necessary for the service to be delivered to 3rd parties [3].

We consider the following actors:

- *Patient*, that depends on the hospital for receiving appropriate health care. Further, patients will refuse to share their data if they do not trust the system or do not have sufficient control over the use of their data;
- *Hospital*, that provides medical treatment and depends on the patients for having their personal information.
- *Clinician*, physician of the hospital that provides medical health advice and, whenever needed, provide accurate medical treatment;
- *Health Care Authority* (HCA) that control and guarantee the fair resources allocation and a good quality of the delivered services.
- *Medical Information System* (MIS), that, according the current privacy legislation, can share the patients medical data if and only if consent is obtained. The *MIS* manages patients information, including information about the medical treatments they have received.

In order to provide rapid and accurate medical treatments, clinicians need a fast access to their patient' medical data. Similarly, HCA needs a fast and reliable access to the data in order to allocate effectively the available resources, and guaranteeing then that each patient can receive a good quality of medical care. Furthermore, HCA wants to be sure that the system cannot be defrauded in any way and that clinicians and patients behave within the limits of their roles. To the other hand, the obvious right of the patient to restrict access on his/her medical data and moreover, to be able to use some safeguards on the privacy of these data, should be taken into serious consideration. The patient's consent must be requested, and he must be notified when its data is shared.

Figure 1 shows the functional requirement model. In the functional requirement model we represent a (Tropos) dependence as an edge labelled by **D**. For every actor we show the goals that they have to aim (**A**) and the services they can offer (**S**). Then we built the trust requirement model. The basic idea is that the owner of an object has full authority concerning access and disposition of his object, and he can also delegate it to other actors. We represent this relationship as an edge labelled by **O**. Further, we want separate the concept of authority from the concept of permission. This allows to use the notion of authority as a prerequisite for creating permissions. By expressing constraints on future delegations one defines the scope of future management structure in an organization. To this end, we introduce the notion of trust and delegation. The basic meaning of trust is to determine whether a actor is authorized to have the object. Thus, we use trust (**T**) to model the basic trust relationship between agents. In the trust management implementation we use delegation to model the actual transfer of rights in some form (e.g. a digital certificate, a signed paper, etc.etc.). We consider two kind

[3] Of course, this is not true for most countries in the world.

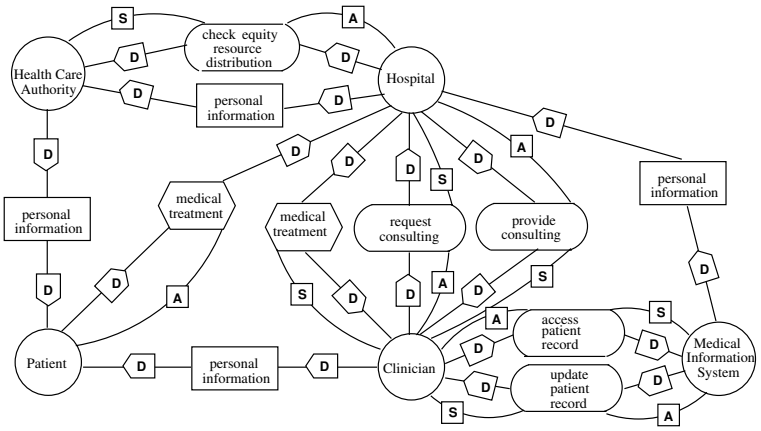

Fig. 1. Health Care System functional requirement model

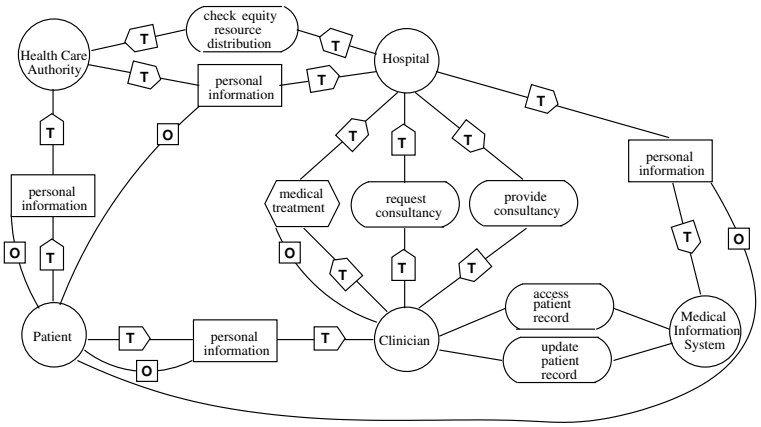

Fig. 2. Health Care System trust requirement model

of delegation: delegation for use, or permission (**P**), and delegation for grant (**G**). We believe that both types of delegation should be offered by an information system since a choice would then result in greater flexibility. Essentially we assume that if an actor is a delegatee for a certain service it must have appropriate permission from the owner of the service. The *Patient* aims to get medical treatments. *Hospital* aims to check equity resource distribution. To the other hand, *Clinician* can offer medical treatments and *HCA* can check equity resource distribution. So, *Patients* depend on the *Hospital* for receiving medical treatments, and in turn, *Hospital* depends on *Clinician* for providing such treatments. *Clinician* depends on the *Patients* for their personal information and on the *Hospital* for specific professional consultancies. The *Hospital* depends on other *Clinicians* for providing professional consultancies and on *HCA* for checking equity resource distribution and for patient personal information. *HCA* depends on *Patient* for personal information. Finally, we also consider the dependencies between *Clinician* and *MIS* to access patient record and to update patient record.

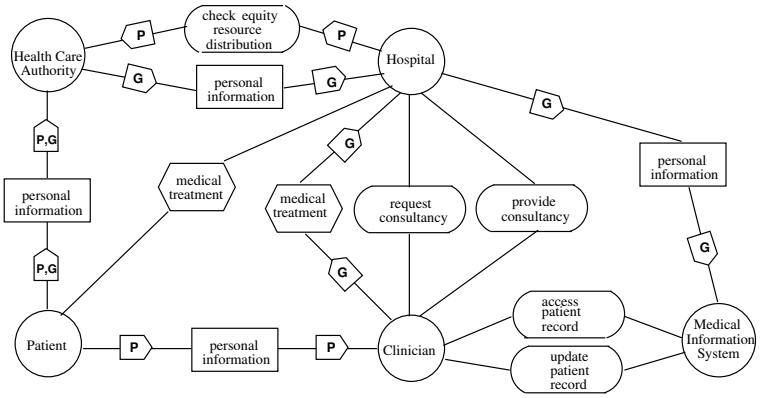

Fig. 3. Health Care System trust management implementation

Figure 2 shows the trust requirement model. The *Patient* owns his personal information and *Clinician* owns medical treatments. *Patient* trusts *HCA* and *Clinician* for his personal information, and *HCA* trusts *Hospital* for it. Further, *Hospital* trusts *HCA* for checking equity resource distribution. *Clinician* trusts *Hospital* for medical treatment and for requesting specific professional consulting, and *Hospital* trusts *Clinician* for providing such consulting. Notice on top of Fig. 2 that there is a trust relationship between two actors (HCA and Hospital) on a resource that is owned by neither of them.

Figure 3 shows the trust management implementation. *Clinician* delegates for grant medical treatments to *Hospital* and *Hospital* delegates for use the goal to check equity resource distribution to *HCA*. *Clinician* and *HCA* need patient personal information to fullfill their service. Thus, *Patient* delegates them his personal information. Further, *HCA* delegates for grant these data to *Hospital*, and in turn, *Hospital* delegates for grant them to *MIS*.

4 Formalization

To built the formal semantics of the requirements model, we use a form of delegation logics to model security requirements. Particularly, we follow Li et al. [16, 17] that provides a logical framework for representing security policies and credentials for authorization in large-scale, open, distributed systems.

We start by presenting the set of predicates for the functional requirement model. When an actor has the capabilities to fulfill a service, he offers it. The intuition is that $\mathsf{offers}(a, s)$ holds if instance a offers the corresponding instance of s. We assume that a can offer the service if he has it. The predicate $\mathsf{aims}(a, s)$ holds if actor a has the objective of reaching to fulfill the goal s. The predicate $\mathsf{depends}(a, b, s1, s2)$ holds if actor a depends from actor b for service $s1$ for the fulfillment of service $s2$.

Next we have predicates for the trust requirement model. The predicate $\mathsf{owns}(a, s)$ holds if the instance a owns the service s. The owner of a service has full authority concerning access and usage of his services, and he can also delegate this authority to other actors. The predicate $\mathsf{trust}(a, b, s1, s2, n)$ holds is actor a trusts actor b for

Functional Requirement Model
$\mathsf{offers}(\texttt{Actor} : a, \texttt{Service} : s)$
$\mathsf{aims}(\texttt{Actor} : a, \texttt{Service} : s)$
$\mathsf{depends}(\texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s1, \texttt{Service} : s2)$
Trust Requirement Model
$\mathsf{owns}(\texttt{Actor} : a, \texttt{Service} : s)$
$\mathsf{trust}\big(\texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s1, \texttt{Service} : s2, \mathcal{N}^+ \cup \{*\} : n\big)$
Trust Management Implementation
$\mathsf{has}(\texttt{Actor} : a, \texttt{Service} : s)$
$\mathsf{fulfills}(\texttt{Actor} : a, \texttt{Service} : s)$
$\mathsf{delGrant}\big(\texttt{id} : idC, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s1, \texttt{Service} : s2, \mathcal{N}^+ \cup \{*\} : n\big)$
$\mathsf{permission}(\texttt{id} : idC, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s1, \texttt{Service} : s2)$

Table 1. Predicates

service $s1$ to fulfill service $s2$; n is called *trust depth*. As suggest by Li at al. [16] for their delegation logics, trust has depth, which is either a positive integer or "*" for unbounded depth.

Finally, we use the following predicates to model the trust management implementation. The basic idea of has is that who has a service, has authority concerning access and disposition of the service, and he can also delegate this authority to other actors provided the owner of the service agree. The predicates fulfills is true when the service is fulfilled by an actor. Particularly, $\mathsf{fulfills}(a, s)$ holds if actor a fulfills the service s. Every trust management framework is based on credentials and delegation. We distinguish two predicates: delGrant and permission. The intuition is that $\mathsf{delGrant}(idC, a, b, s1, s2, n)$ holds if actor a delegates the permission to grant the service $s1$ to fulfill the goal $s2$ to actor b. The intuition is that $\mathsf{permission}(idC, a, b, s1, s2)$ holds if actor a delegates the permission to use the service $s1$ to reach the goal $s2$ to actor b. The actor a is called the *delegater*; the actor b is called the *delegatee*; idC is the certificate identify; n is called the *delegation depth*. A delegation has depth as for trust. One way to view depth is the number of re-delegation steps that are allowed; depth 1 means that no re-delegation is allowed, depth N means that $N-1$ further step are allowed, and depth "*" means that unbounded re-delegation is allowed. We abbreviate delegation and permission chain as follows

$$\mathsf{delGChain}(A, B, S1, S2) \equiv \begin{cases} \exists k \text{ s.t. } \exists a_1 \dots a_k \ \exists n_1 \dots n_{k-1} \ \forall i \in [1 \dots k-1] \\ \mathsf{delGrant}(id_i, a_i, a_{i+1}, S1, S2, n_i) \wedge \ a_1 = A \wedge a_k = B \end{cases}$$

$$\mathsf{permissionChain}(A, C, S1, S2) \equiv \begin{cases} (\mathsf{permission}(idC, A, C, S1, S2)) \vee \\ (\exists B \ \mathsf{delGChain}(A, B, S1, S2) \ \wedge \\ \mathsf{permission}(idC, B, C, S1, S2)) \end{cases}$$

5 Axioms and Properties

In order to illustrate our approach we formalize the case study and check-model it in Datalog [1]. A datalog program is a set of rules of the form L:- $L_1 \wedge ... \wedge L_n$ where

	Functional Requirement Model
Ax1:	$\mathsf{aims}(B, S1)$:- $\mathsf{depends}(A, B, S1, S2)$
	Trust Requirement Model
Ax2:	$\mathsf{trust}(A, B, S1, S2, N-1)$:- $\mathsf{trust}(A, B, S1, S2, N) \wedge N > 2$
Ax3:	$\mathsf{trust}(A, C, S1, S2, P)$:- $\mathsf{trust}(A, B, S1, S2, N) \wedge \mathsf{trust}(B, C, S1, S2, M) \wedge P = \min\{N-1, M\} \wedge N > 2$
	Trust Management Implementation
Ax4:	$\mathsf{has}(A, S)$:- $\mathsf{owns}(A, S)$
Ax5:	$\mathsf{has}(B, S1)$:- $\mathsf{delGrant}(ID, A, B, S1, S2, N)$
Ax6:	$\mathsf{has}(B, S1)$:- $\mathsf{permission}(ID, A, B, S1, S2)$
Ax7:	$\mathsf{fulfills}(A, S)$:- $\mathsf{has}(A, S) \wedge \mathsf{offers}(A, S)$
Ax8:	$\mathsf{fulfills}(A, S1)$:- $\mathsf{depends}(A, B, S1, S2) \wedge \mathsf{fulfills}(B, S1)$
Ax9:	$\mathsf{fulfills}(A, S)$:- $\forall S' \sqsubseteq S$, $\mathsf{fulfills}(A, S')$

Table 2. Axioms

L, called head, is a positive literal and $L_1, ..., L_n$ are literals and they are called body. Intuitively, if $L_1, ..., L_n$ are true in the model then L must be true in the model. In Datalog, negation is treated as negation as failure. In other words, if there is no evidence that an atom is true, it is considered to be false, and hence if an atom is not true in some model, then its negation should be considered to be true in that model. In this way, if a subgoal is not fulfilled, also the correspondent main goal is not fulfilled.

The intuitive descriptions of systems are often incomplete, and need to be completed for a correct analysis. To draw the right conclusions from an intuitive model, we need to complete the model using a systematic method. To this end we use *axioms*.

In Table 2 we present the axioms for our framework. Ax1 says that if an actor depends to other actors to fulfill a service the last has as objective the service. Ax2 states that if someone trust with depth N, then he trust with smaller depth. Ax3 completes the trust relationship between actors. As we say before, the owner of a service has full authority concerning access and disposition of it. Thus, Ax4 states that if an actor owns a service, he has it. Ax5 and Ax6 say that the delegatee has the service. Ax7 states that if an actor has a service and offers it, then he fulfills the service. Ax8 says that if an actor depends to another and the second fulfills the service, also the first fulfill the service. Ax9 is for and-decomposition and states that an actor fulfills the main service if he has fulfilled all its subservices. For or-decomposition the main goal is fulfilled if one of its subgoals is fulfilled. Note that $S' \sqsubseteq S$ means that S' is subgoal of S.

Properties are different from axioms: they are design feature that must be checked. If the set of features is not consistent, i.e. they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. In Table 3 we use the $A \Rightarrow? B$ to mean that one must check that each time A holds it is desirable that B also holds. In Datalog this can be rapresented as the constraint :- $A,$ not B.

Table 3 shows a number of properties. Pro1 wants to check if an actor fulfills the services that he has as objective. Pro2 and Prop3 state that if an actor has or fulfills a service and it belongs to another actor, the last has to trust first one. Pro4 and Prop5 state that if an actor has or fulfills a service and it belongs to another actor, there is a delegation chain from the first to the second. Pro6, Pro7, and Pro8 state that if an agent

Pro1:	$\mathsf{aims}(A,S) \Rightarrow? \mathsf{fulfills}(A,S)$
Pro2:	$\mathsf{has}(B,S1) \wedge \mathsf{owns}(A,S1) \wedge A \neq B \Rightarrow? \exists N\ \mathsf{trust}(A,B,S1,S2,N)$
Pro3:	$\mathsf{fulfills}(B,S1) \wedge \mathsf{owns}(A,S1) \wedge A \neq B \Rightarrow? \exists N\ \mathsf{trust}(A,B,S1,S2,N)$
Pro4:	$\mathsf{has}(B,S1) \wedge \mathsf{owns}(A,S1) \wedge A \neq B \Rightarrow? \begin{cases} \mathsf{delGChain}(A,B,S1,S2) \vee \\ \mathsf{permissionChain}(A,B,S1,S2) \end{cases}$
Pro5:	$\mathsf{fulfills}(B,S1) \wedge \mathsf{owns}(A,S1) \wedge A \neq B \Rightarrow? \mathsf{permissionChain}(A,B,S1,S2)$
Pro6:	$\mathsf{fulfills}(A,S) \Rightarrow? \mathsf{has}(A,S)$
Pro7:	$\mathsf{permission}(ID,A,B,S1,S2) \Rightarrow? \mathsf{has}(A,S1)$
Pro8:	$\mathsf{delGrant}(ID,A,B,S1,S2,N) \Rightarrow? \mathsf{has}(A,S1)$
Pro9:	$\mathsf{permission}(ID,A,B,S1,S2) \Rightarrow? \exists N\ \mathsf{trust}(A,B,S1,S2,N)$
Pro10:	$\mathsf{delGrant}(ID,A,B,S1,S2,N) \Rightarrow? \exists M \geq N\ \mathsf{trust}(A,B,S1,S2,M)$
Pro11:	$\mathsf{permissionChain}(A,B,S1,S2) \Rightarrow? \exists N\ \mathsf{trust}(A,B,S1,S2,N)$
Pro12:	$\mathsf{delGChain}(A,B,S1,S2) \Rightarrow? \exists N\ \mathsf{trust}(A,B,S1,S2,N)$
Pro13:	$\mathsf{delGChain}(A,B,S1,S2) \Rightarrow? \begin{cases} \exists M\ \ \exists A_1 \ldots A_M\ \ \exists N_1 \ldots N_{M-1} \\ \forall i \in [1 \ldots M-1] \\ \mathsf{delGrant}(ID_i, A_i, A_{i+1}, S1, S2, N_i) \wedge \\ A_1 = A \wedge A_M = B \wedge N_i > N_{i+1} \end{cases}$

Table 3. Desirable Properties of a Design

fulfills or delegates a service, he should have it. Pro9, Pro10, Pro11, and Pro12 state that an actor who delegates something to other or there is a delegation chain, the delegater has to trust the delegatee. Rights or privileges can be given to trusted agents that are then responsible for agents they may delegate this right to. So the agents will only delegate to agents that they trust. This forms a delegation chain. If any agent along this chain fails to meet the requirements associated with a delegated right, the chain is broken and all agents following the failure are not permitted to perform the action associated with the right. Thus, Prop13 is used to verify whether the delegate chain is valid.

As already proposed in [12], our framework supports automatic verification of security requirements. Particularly, we use the DLV system [9] to check system consistency.

6 Negative Authorizations

In all practical example of policies and requirements for e-health we found the need for negative authorization (for non-functional requirements) and negative goals or goal whose fulfillment obstacles the fulfillment of other goals (for functional requirements). Tropos already accommodates the notion of positive or negative contribution of goals to the fulfillment of other goals. We only need to lift the framework to permission and trust. Notice that having negative authorization in the requirements model does *not* mean that we must use "negative" certificates. Even if some form of negative certificates are often used in real life[4] we can use negative authorization to help the designer in shaping the perimeter of positive trust, i.e. positive certificates, to avoid incautious delegation certificates that may give more powers than desired.

[4] E.g., A certificate issued by the government that you have no pending criminal trials

Trust Management Implementation
$\mathsf{delDenial}\big(\mathtt{id} : idC, \mathtt{Actor} : a, \mathtt{Actor} : b, \mathtt{Service} : s, \mathcal{N}^+ \cup \{*\} : n\big)$
$\mathsf{prohibition}(\mathtt{id} : idC, \mathtt{Actor} : a, \mathtt{Actor} : b, \mathtt{Service} : s)$

Table 4. Negative Authorization Predicates

We use a *closed world* policy. Under this policy, the lack of an authorization is interpreted as a negative authorization. Therefore, whenever a actor tries to access an object, if a positive authorization is not found in the system, the actor is denied the access. This approach has a major problem in the lack of a given authorization for a given actor does not prevent this user from receiving this authorization later on.

Suppose that an actor should not be given access to a service. In situations where authorization administration is decentralized, an actor possessing the right to use the service, can delegate the authorization on that service to the wrong actor. Since many actors may have the right to use a service, it is not always possible to enforce with certainty the constraint that a actor cannot access a particular service. We propose an explicit *negative* authorization as an approach for handling this type of constraint.

An explicit negative authorization express a *denial* for an actor to access a service. In our approach negative authorizations are stronger than positive authorizations. That is, whenever a user has both a positive and a negative authorization on the same object, the user is prevented from accessing the object. Negative authorizations in our model are handled as blocking authorizations. Whenever a user receives a negative authorization, his positive authorizations become blocked. We distinguish two predicates: $\mathsf{delDenial}$ and $\mathsf{prohibition}$. The intuition is that $\mathsf{delDenial}(idC, a, b, s, n)$ holds if actor a delegates the permission to denial the service s to actor b. The intuition is that $\mathsf{prohibition}(idC, a, b, s)$ holds if actor a forbids to use the service s to actor b. Actor a says that service s cannot be assigned to actor b We assume that if an actor a denial an actor b to have service s there is not a delegation chain from a to b. So, if a is the owner of s then b cannot have s. Otherwise b could have s if there exists a delegation chain from owner of s and b without a. As done for positive authorization, we can define an abbreviation for a denial chain as

$$\mathsf{delDChain}(A, B, S) \equiv \begin{cases} \exists k \text{ s.t. } \exists a_1 \dots a_k \ \exists n_1 \dots n_{k-1} \ \forall i \in [1 \dots k-1] \\ \mathsf{delDenial}(id_i, a_i, a_{i+1}, S, n_i) \wedge \ a_1 = A \wedge a_k = B \end{cases}$$

and an abbreviation for a prohibition chain as

$$\mathsf{prohibitionChain}(A, C, S) \equiv \begin{cases} (\mathsf{prohibition}(idC, A, C, S)) \vee \\ (\exists B \ \mathsf{delDChain}(A, B, S) \ \wedge \mathsf{prohibition}(idC, B, C, S)) \end{cases}$$

As we say for positive authorization, the intuitive description of systems are often incomplete, and need to be completed for providing correct analysis. To this end we use axioms to complete the model. In Table 5 we show how. Ax5 and Ax6 are modified to account for the possibility of negative authorizations: we have to add in the body of the rules that there is not a prohibition chain from the delegater to the delegatee.

Trust Management Implementation
Ax5: $\mathsf{has}(B, S1)$:- $\mathsf{delGrant}(ID, A, B, S1, S2, N) \wedge$ not $\mathsf{prohibitionChain}(A, B, S1)$
Ax6: $\mathsf{has}(B, S1)$:- $\mathsf{permission}(ID, A, B, S1, S2) \wedge$ not $\mathsf{prohibitionChain}(A, B, S1)$

Table 5. Negative Authorization Axioms

Pro14: $\mathsf{prohibition}(ID, A, B, S) \wedge \mathsf{owns}(A, S) \Rightarrow?$ not $\mathsf{has}(B, S)$
Pro15: $\mathsf{prohibition}(ID, A, B, S) \wedge \mathsf{owns}(A, S) \Rightarrow?$ not $\mathsf{fulfills}(B, S)$
Pro16: $\mathsf{delDChain}(A, B, S) \Rightarrow? \begin{cases} \exists M \ \exists A_1 \dots A_M \ \exists N_1 \dots N_{M-1} \\ \forall i \in [1 \dots M-1] \\ \mathsf{delDenial}(ID_i, A_i, A_{i+1}, S, N_i) \wedge \\ A_1 = A \wedge A_M = B \wedge N_i > N_{i+1} \end{cases}$

Table 6. Negative Authorization Desirable Properties

System designer should check that the system is secure. Table 6 presents properties for negative authorization to verify if the model respects security features. Pro14 and Pro15 check that if the owner of a service forbids to use it to another actor, the last one cannot have and fulfill the service. Pro16 is used to verify if a denial chain is valid.

7 Trust Management Implementation

Several trust management systems [5, 4, 16] have been proposed to address authorization in decentralized environments. In this section we present the implementation of our approach using the RT (Role-based Trust-management) framework [17–19].

The RT framework provides policy language, semantics, deduction engine, and pragmatic features such as application domain specification documents that help distributed users maintain consistent use of policy terms. In comparison with systems like SPKI/SDSI [10, 24] and KeyNote [4], the advantage of RT includes a declarative, logic-based semantic foundation also based on Datalog, support for vocabulary agreement, strongly-typed credential and policies, and more flexible delegation structures.

An entity in RT is a uniquely identified individual or process. They can issue credentials and make requests and we assume that one can determine which entity issued a particular credential or request. RT uses the notion of roles to represents attributes: an entity is a member of a role if and only if it has the attribute identified by the role.

In RT, a role is denoted by an entity followed by a role name, separated by a dot. Only the entity A has the authority to define the members of the role $A.R$, and A does so by issuing role-definition credentials. An entity A can define $A.R$ to contain $A.R_1$, another role defined by A. Such a credential reads $A.R \longleftarrow A.R_1$; it means that A defines that R_1 dominates R. At the same time, a credential $A.R \longleftarrow B.R$ is a delegation from A to B of authority over R. This can be used to decentralize the user-role assignment. A credential of the form $A.R \longleftarrow B.R_1$ can be used to define role-mapping across multiple organizations when they collaborate; it also represents a delegation from A to B. Using a linked role in a credential enables the issuer to delegate

to each member of a role. The credential $A.R \longleftarrow A.R_1.R_2$ states that: $A.R$ contains any $B.R_2$ if $A.R_1$ contains B.

To model permissions on objects and services one also uses roles. A permission typically consists of an access mode and an object. It is often useful to group logically related objects and access modes together and to give permission about them together. These groups are called o-set and are defined in ways similar to roles. The difference is that the members of o-set are objects that are not entities.

A o-set-definition credential is similar to a role-definition credential.

- $A.o(h_1, ..., h_n) \longleftarrow B.o_1(s_1, ..., s_m)$
  where $o(h_1, ..., h_n)$ is an o-set name of base type τ, and $B.o_1(s_1, ..., s_m)$ another o-set of base type τ.
- $A.o(h_1, ..., h_n) \longleftarrow A.r_1(t_1, ..., t_l).o_1(s_1, ..., s_m)$
  where $o(h_1, ..., h_n)$ is an o-set name of base type τ, and $A.r_1(t_1, ..., t_l).o_1(s_1, ..., s_m)$ is a linked o-set in which $r_1(t_1, ..., t_l)$ is a role name and $o_1(s_1, ..., s_m)$ is an o-set name of base type τ.

At present we do not have roles in our framework (through roles are present in the Tropos framework), and so, we do not translate role-definition credentials. The intuition is that $\mathsf{permission}(ID, A, B, S1, S2)$ can be rewritten in RT framework as $A.S1 \longleftarrow B.S2$, and $\mathsf{delGrant}(ID, A, B, S1, S2, N)$ as $A.S1 \longleftarrow B.r.S2$, where B allows to use the service $S1$ to actors in the role $B.r$.

The user of the system - patients, clinicians and administrative staff - are modeled as entity whose policies consists only of credentials they acquire over time.

Example 1. A patient allows his clinician to read his personal/medical data to provide accurate medical treatment. We express the trust relationship in our framework as $\mathsf{permission}(id, Pat, Cli, Rec, MedTre)$:- $\mathsf{isClinicianOf}(Pat, Cli) \wedge \mathsf{owns}(Pat, Rec)$ The intuition is that $\mathsf{isClinicianOf}(a, b)$ holds if the instance a is the clinician of the instance b. Now, we translate the relationship into the RT framework as

$\mathsf{Pat.recordAc}(\mathsf{read}, ?\mathsf{F} : \mathsf{Pat.record}) \longleftarrow \mathsf{Pat.clinician.provide}(?\mathsf{E} : \mathsf{medTre})$

Given "Pat.record ⟵ Rec" and "Pat.clinician ⟵ Cli", one can conclude that "$\mathsf{Pat.recordAc}(\mathsf{read}, \mathsf{Rec}) \longleftarrow \mathsf{Cli.provide}(?\mathsf{E} : \mathsf{medTre})$".

Example 2. The Medical Information System allows the clinician to write on his patient records to upgrade them. We express the trust relationship as $\mathsf{permission}(id, MIS, Cli, Rec, upgrade(Rec))$:- $\mathsf{isClinicianOf}(Pat, Cli) \wedge \mathsf{owns}(Pat, Rec)$

Now, we translate the relationship into the RT framework as

$\mathsf{MIS.recordAc}(\mathsf{write}, ?\mathsf{F} : \mathsf{Pat.record}) \longleftarrow \mathsf{Pat.clinician.upgrade}(?\mathsf{F} : \mathsf{Pat.record})$

Given "Pat.record ⟵ Rec" and "Pat.clinician ⟵ Cli", one can conclude that "$\mathsf{MIS.recordAc}(\mathsf{write}, \mathsf{Rec}) \longleftarrow \mathsf{Cli.upgrade}(\mathsf{Rec})$".

8 Conclusion

The main contribution of this paper is the introduction of a framework that integrates security and requirements engineering. We have proposed a PKI/trust management requirements specification and analysis framework based on the clear separation of trust and delegation relationship. This distinction makes possible to capture the high-level security requirements without being immediately bogged down into considerations about cryptographic algorithms or security implementation. This should be similar to what happens when discussing functional system requirements: one doesn't get immediately trapped into discussions about programming languages or Java patterns and coding techniques.

Further, the framework we proposed supports the automatic verification of security requirements specified in a formal modeling language. Particularly, we have used the DLV system to check system consistency. Finally, we have defined the trust management implementation of our framework into the RT framework.

The research developed here is still in progress. Much remains to be done to further refine the proposed framework and validate its usefulness with real case studies. We are currently working in the direction of incorporating explicitly roles adding time features and the integration with the Formal Tropos tool [11]. Also we are investigating the effects of supporting hierarchies of objects and hierarchies of actors.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.
3. A. Bandara, E. Lupu, and A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. In *Proc. of the 4th Int. Workshop on Policies for Distributed Sys. and Networks (POLICY'03)*, 2003.
4. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The Role of Trust Management in Distributed Systems Security. *Secure Internet Programming*, 1603:185–210, 1999.
5. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of 1996 IEEE Symp. on Sec. and Privacy*, pages 164–173, 1996.
6. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.
7. Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.
8. N. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, University of London, 2002.
9. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. of IJCAI'03*. Morgan Kaufmann Publishers, 2003.
10. C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple Public Key Certificates. Internet Draft (work in progress), 1999.
11. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements: Some experimental results. In *Proc. of RE'03*, page 105. IEEE Press, 2003.

12. P. Giorgini, F. Massacci, J. Mylopoulous, and N. Zannone. Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning. In *Proc. of iTrust 2004*, volume 2995 of *LNCS*, pages 176–190. Springer-Verlag Heidelberg, 2004.
13. S. Guttorm. Eliciting Security Requirements by Misuse Cases. In *Proc. of TOOLS Pacific 2000*, 2000.
14. T. Jim. SD3: a trust management system with certified evaluation. In *Proc. of 2001 IEEE Symp. on Sec. and Privacy*, pages 106 – 115. IEEE Press, 2001.
15. J. Jürjens. Towards Secure Systems Development with UMLsec. In *Proc. of FASE'01*, 2001.
16. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *TISSEC*, 6(1):128–171, 2003.
17. N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust-management Languages. In *Proc. of PADL'03*, 2003.
18. N. Li, J. C. Mitchell, and W. H. Winsborough. Design of A Role-based Trust-management Framework. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, 2002.
19. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed Credential Chain Discovery in Trust Management. *J. of Comp. Sec.*, 11(1):35–86, 2003.
20. L. Liu, E. S. K. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of RE'03*, pages 151–161, 2003.
21. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *Proc. of UML'02*, volume 2460, pages 426–441. Springer-Verlag Heidelberg, 2002.
22. J. Lopez, A. Mana, J. A. Montenegro, and J. J. Ortega. PKI design based on the use of on-line certification authorities. *Int. J. of Information Sec.*, 2(2):91 – 102, 2004.
23. J. McDermott and C. Fox. Using Abuse Care Models for Security Requirements Analysis. In *Proc. of ACSAC'99*, 1999.
24. R. L. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure, 1996.
25. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.
26. T. Tryfonas, E. Kiountouzis, and A. Poulymenakou. Embedding security practices in contemporary information systems development approaches. *Inform. Management and Comp. Sec.*, 9:183–197, 2001.