

ANALYSING SECURITY REQUIREMENTS OF INFORMATION SYSTEMS USING TROPOS

Haralambos Mouratidis

Innovative Informatics Group, School of Computing and Technology, Univ. of East London
haris@uel.ac.uk

Abstract. Security is an important issue when developing complex information systems, however very little work has been done in integrating security concerns during the analysis of information systems. Current methodologies fail to adequately integrate security and systems engineering, basically because they lack concepts and models as well as a systematic approach towards security. We believe that security should be considered during the whole development process and it should be defined together with the requirements specification. This paper introduces extensions to the Tropos methodology to accommodate security. A description of new concepts is given along with an explanation of how these concepts are integrated to the current stages of Tropos. The above is illustrated using an agent-based health and social care information system as a case study.

1 INTRODUCTION

Analysis is one of the most important stages in the whole software engineering process. This is because if the analysis of the system is wrong all the following stages will end up wrong. It is very important during the analysis stage that the software engineer understands exactly the problem that they have to tackle. This can be a very difficult process especially if the system is new and there is no previous version of a computer system to serve as a model. To understand the problem, the software engineer must understand the user needs and requirements. Concepts and languages for analysis are needed to deal with the system as a whole with organisational and coordination properties, as well as the individual components of the system and their properties.

Tropos (Castro, 2001) is an information system development methodology, tailored to describe both the organisational environment of a system and the system itself, employing the same concepts throughout the development stages. *Tropos* adopts the *i** modelling framework (Yu, 1995), which uses the concepts of actors, goals, soft goals, tasks, resources and social dependencies for defining the obligations of actors (dependees) to other actors (dependers). Actors have strategic goals and intentions within the system or the organisation and represent (social) agents (organisational, human or

software), roles or positions (represents a set of roles). A goal represents the strategic interests of an actor. In Tropos we differentiate between hard (only goals hereafter) and soft goals. The latter having no clear definition or criteria for deciding whether they are satisfied or not. A task represents a way of doing something. Thus, for example a task can be executed in order to satisfy a goal. A resource represents a physical or an informational entity while a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource.

One distinctive characteristic of Tropos is the fact that it covers the very early phases of requirements analysis. This allows for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human users. By considering early phases of the requirements analysis, the main advantage is that 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* covers four main software development phases:

Early Requirements, concerned with the understanding of a problem by studying an existing organisational setting; the output of this phase is an organisational model, which includes relevant actors and their respective dependencies;

Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities; this description models the system as a (small) number of actors which have a number of dependencies with actors in their environment; these dependencies define the system's functional and non-functional requirements;

Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows; within the framework, subsystems are represented as actors and data/control interconnections are represented as (system) actor dependencies;

Detailed design, where each architectural component is defined in further detail in terms of inputs, outputs, control, and other relevant information. *Tropos* is using elements of UML (Jacobson, 1999) to complement the features of *i**.

In addition to the graphical representation, *Tropos* provides a formal specification language called Formal *Tropos* (Fuxman, 2001).

Although *Tropos* can partially model security concerns (Mouratidis, 2002 – Yu, 2002), it has not conceived with security in mind and it does not provide models and notations to adequately model security aspects. Security is an important issue when developing complex information systems and the lack of models and notation to capture it, restricts the usefulness of a development methodology.

However, so far very little work has taken place in integrating security and systems engineering. The common approach towards the inclusion of security within a system is to identify security requirements after the definition of a system. This approach has provoked the emergence of computer systems afflicted with security vulnerabilities (Stallings, 1999). From the viewpoint of the traditional security paradigm, it should be possible to eliminate such problems through more extensive use of formal methods and better software engineering.

We believe that security should be considered during the whole development process and it should be defined together with the requirements specification. By considering security only in certain stages of the development process, more likely, security needs will conflict with functional requirements of the system. Taking security into account along with the functional requirements throughout the development stages helps to limit the cases of conflict, by identifying them very early in the system development, and find ways to overcome them. On the other hand, adding security as an afterthought not only increases the chances of such a conflict to exist, but it requires huge amount of money and valuable time to overcome it, once they

have been identified (usually a major rebuild of the system is needed).

This paper introduces extensions to *Tropos*, to accommodate security concerns during the early requirements analysis. The proposed extensions are illustrated with the aid of a case study, the electronic Single Assessment Process (eSAP) system, a real-life integrated health and social care information system for older people (Mouratidis, 2002b). The eSAP project is a joint research project, between the Computer Science Department and the Sheffield Institute for Studies on Ageing (SISA), both at the University of Sheffield, and it aims to deliver the Single Assessment Process, a national policy in England of an integrated assessment of health and social care needs of older people [www.doh.gov.uk/scg/sap/].

This paper is structured as follows. Section 2 presents security concerns in software engineering. Section 3 introduces an extension to the *Tropos* methodology in order to accommodate security and in Section 4 such an extension is applied to the eSAP case study. Finally, Section 5 summarizes the contributions of the paper and points to further work.

2 SOFTWARE ENGINEERING AND SECURITY

Software engineering considers security requirements, as well as performance and reliability requirements, as non-functional requirements. Non-functional requirements represent the constraints under which the system must operate but also introduce quality characteristics to the system. Software designers have already recognised the importance of integrating non-functional requirements, such as performance and reliability, into software design processes (Lampson, 2000) however security requirements are still an afterthought.

This typically means that security enforcement mechanisms have to be fitted into a pre-existing design therefore leading to serious design challenges, which usually translate into software vulnerabilities. Modern computer systems, applications and operating systems are full of security vulnerabilities in many levels therefore leading to the violation of the security policy. Adopting a security focus through the overall system development process represents a solution to mitigate such problems.

There are at least two reasons for the lack of support for security engineering (Meadows, 1994).

The first reason is that security requirements are generally difficult to analyse and model. A second important reason is lack of developer acceptance and expertise for secure software development. For software developers, security interferes with features and time to market. Furthermore security policies are generally specified in terms of security models that are not integrated with general software engineering models.

A major problem in analysing non-functional requirements is that there is a need to separate functional and non-functional requirements yet, at the same time, individual non-functional requirements may relate to one or more functional requirements. If the non-functional requirements are stated separately from the functional requirements, it is sometimes difficult to see the correspondence between them. If stated with the functional requirements, it may be difficult to separate functional and non-functional considerations.

However, security is an important aspect in the development of complex computerised systems and according to Devanbu (Devanbu, 2000) "Security concerns must inform every phase of software development, from requirements engineering to design, implementation, testing and deployment". The consideration of security in early software development stages will aid in the elimination of security vulnerabilities that are difficult and expensive to correct during later stages.

3 MODELING SECURITY WITH TROPOS

As mentioned above, Tropos has not been conceived with security in mind. Thus, we have extended Tropos, introducing concepts such as security constraint, secure dependency, and secure goal/ task/ resource in order to provide a systematic process that will guide the developer in considering security requirements during the whole development phases.

Basically, security analysis consists of analysing the security needs in terms of *security constraints*, imposed to the system and the stakeholders, and of the identification of *secure entities* that can guarantee the satisfaction of such constraints. The analysis allows also for the identification of capabilities of the system in order to help towards the satisfaction of the *secure entities*. In this work, we focus mainly in the integration of security analysis in the early requirement stage of the *Tropos* methodology.

3.1 Security concepts

Constraints can be categorised according to the non-functional requirement they are related to (e.g., reliability, performance or security constraints). In this work we are interested in imposing to the system constraints that help towards the security of the system. We define *security constraint* as a constraint that is related to the security of the system.

In the early requirements analysis security constraints are identified and analysed according to the constraint analysis processes we have proposed in (Mouratidis – 2002c). *Security constraints* are then imposed to different parts of the system, and possible conflicts between security and other (functional and non functional) requirements of the system are identified and solved. It is worth mentioning that we consider a *security constraint* contributing to a higher level of abstraction, meaning that a *security constraint* does not involve the identification of particular security protocols so that it does not restrict the development of the system to a specific security solution. This means we are not taking into consideration specific security protocols that should be decided during the implementation of the system, and that most of the times restrict the design with the use of a particular implementation language.

A *security constraint* is represented graphically as shown in figure 1.



Figure 1. A graphical representation of a security constraint

The term *secure entities* involves any *secure goals, tasks and resources* of the system. A *secure entity* is introduced to the actor (or the system) in order to help in the achievement of a *security constraint*. A *secure goal* does not particularly define how the *security constraint* can be achieved, since (as in the definition of goal, see (Yu, 1995)) alternatives can be considered. However, this is possible through a *secure task*, since a task specifies a way of doing something (Yu, 1995). Thus, a *secure task* represents a particular way for satisfying a *secure goal*. A resource that is related to a *secure entity* or a *security constraint* is considered a *secure*

resource. *Secure Entities* are represented graphically by introducing an S within brackets (S) before the text description as shown in figure 2.



Figure 2. Graphical representation of secure entities (task, goal, and resource, respectively)

A *secure dependency* introduces *security constraint(s)*, proposed either by the depender (most likely) or the dependee (most unlikely) in order to successfully satisfy the dependency. For example a *Doctor* (depender) depends on a *Patient* (dependee) to obtain *Health Information* (dependum). However, the *Patient* imposes a *security constraint* to the *Doctor* to share *health information only if consent is achieved*. Both the depender and the dependee must agree in this constraint (or constraints) for the secure dependency to be valid. That means, in the depender side, the depender expects from the dependee to satisfy the *security constraints* while in the dependee side, a secure dependency means that the dependee will make an effort to deliver the dependum by satisfying the *security constraint(s)*. There are two degrees of security: *Open Secure dependency* (normal dependency) and *Secure dependency*. In an *Open Secure Dependency* some security conditions might be introduced but if the dependee fail to satisfy them, the consequences will not be serious. This means that the security of the system will not be in danger if some of these conditions are not satisfied. An *Open Secure Dependency* is graphically represented (Figure 3-a) as unmarked (as the normal dependency). On the other side, there are three different types of a secure dependency:

- *Dependee Secure Dependency*, depender depends on dependee and dependee introduces security constraints for the dependency. Depender must satisfy the security constraints introduced by the dependee in order to help in the achievement of the secure dependency. This type of secure dependency is graphically represented with a constraint at the side of the depender (Figure 3-b).
- *Depender Secure Dependency*, depender depends on dependee, and depender introduces security constraints for the dependency. The dependee must satisfy the security constraints introduced by the depender, otherwise the security of the dependency will be in risk. This type of secure dependency is graphically

represented with a constraint at the side of the dependee (Figure 3-c).

- *Double Secure Dependency*, depender depends on dependee and both depender and dependee introduce security constraints for the dependency. Both must satisfy the security constraints introduced to achieve the secure dependency. This type of secure dependency is represented with constraints on both sides (Figure 3-d).

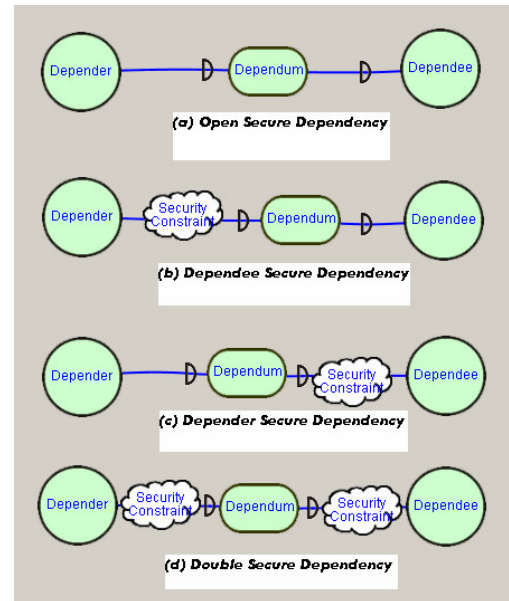


Figure 3. The Different Types of Secure Dependencies

3.2 Formal Tropos

Formal Tropos (Fuxman, 2001) complements graphical Tropos by extending the Tropos graphical language into a formal specification language (Fuxman, 2001). The language offers all the primitive concepts of graphical Tropos, supplemented with a rich temporal specification language, inspired by KAOS (Dardenne, 1993), that has formal semantics and it is amenable to formal analysis. In addition, Formal Tropos offers a textual notation for *i** models and allows the description of different elements of the specification in a first order linear-time temporal logic. A specification of formal Tropos consists of a sequence of declarations of entities, actors, and dependencies (Fuxman, 2001).

Formal Tropos can be used to perform a formal analysis of the system and also verify the model of

the system by employing formal verification techniques such as model checking to allow for an automatic verification of the system properties (Fuxman, 2001).

As with the graphical Tropos, Formal Tropos has not been conceived with security on mind. Thus we felt that extending Formal Tropos was the next natural step in our security extensions for two reasons. Firstly, formal Tropos fails to adequately model some security aspects (such as secure dependencies and security constraints), and secondly formal Tropos allows the formal analysis of our introduced concepts and thus provides formalism to our approach. Towards this direction, we have extended Formal Tropos grammar as shown below (**bold** letters indicate the extensions).

entity:= Entity name [attributes] [creation-properties] [invar-properties] [**security-properties**]

actor:= Actor name [attributes] [creation-properties] [invar-properties] [**security-properties**] [actor-goals]

dependency:= Dependency name type **security-type** mode Depender name Dependee name [attributes] [creation-properties] [invar-properties] [fulfil-properties] [**security-properties**]

security-type:= security type (Depender |Dependee |Double |Open Secure)

security-properties:= Security security-property*

security-property:= Constraint property-origin temporal-formula

4 THE ESAP EXAMPLE

In the early requirements analysis the goals and the dependencies between the stakeholders (actors) are modelled. For this purpose *Tropos* introduces actor diagrams. In such a diagram each node represents an actor, and the links between the different actors indicate that one depends on the other to accomplish some goals. In addition, *security constraints* are imposed to the stakeholders of the system (by other stakeholders). These constraints are analysed and *security entities* are introduced.

In the case of the eSAP system, we have four actors (Figure 4):

- *Older Person*: The Older Person that wishes to receive appropriate health and social care (patient)
- *Professional*: The health and/or social care professional
- *DoH*: The English Department of Health
- *Benefits Agency*: An agency that helps the older person financially

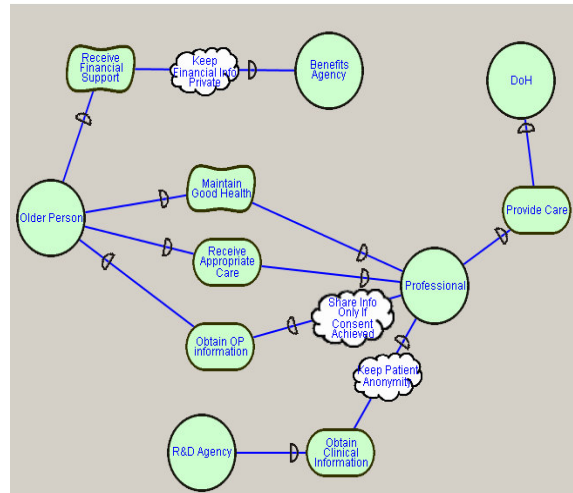


Figure 4. Actor diagram of the eSAP including security constraints

Figure 4 illustrates part of the actor diagram of the eSAP system taking into consideration security constraints that are imposed to the stakeholders of the system.

The *Older Person* depends on the *Benefits Agency* to *Receive Financial Support*. However, the *Older Person* worries about the privacy of their finances so they impose a constraint to the *Benefits Agency* actor, to keep their financial information private. The *Professional* depends on the *Older Person* to *Obtain Information*, however one of the most important and delicate matters for a patient (in our case the older person) is the privacy of their personal medical information, and the sharing of it. Thus most of the times the *Professional* is imposed a constraint to share this information if and only if consent is achieved. In addition, one of the main goals of the *R&D Agency* is to *Obtain Clinical Information* in order to perform tests and research. To get this information the *R&D Agency* depends on the *Professional*. However, the *Professional* is imposed a constraint (by the Department of Health) to *Keep Patient Anonymity*.

In addition, the security constraints imposed at each actor are further analysed by identifying which goals of the actor they restrict (Figure 5). The assignment of a *security constraint* to a goal is indicated using a constraint link (a link that has the “restricts” tag). For example, the *Professional* actor has been imposed two *security constraints* (*Share Info Only If Consent Achieved* and *Keep Patient Anonymity*). During the means-end analysis of the *Professional* actor we have identified the *Share Medical Info* goal. However, this goal is restricted by the *Share Info Only If Consent Achieved* constraint imposed to the *Professional* by the *Older Person*. For the *Professional* to satisfy the constraint, a secure goal is introduced *Obtain Older Person Consent*. However this goal can be achieved with many different ways, for example a *Professional* can obtain the consent personally or can ask a nurse to obtain the consent on their behalf. Thus a sub-constraint is introduced, *Only Obtain Consent Personally*. This sub constraint introduces another *secure goal Personally Obtain Consent*. This goal is divided into two sub-tasks *Obtain Consent by Mail* or *Obtain Consent by Phone*.

The *Professional* has also a goal to *Provide Medical Information for Research*. However, the constraint *Keep Patient Anonymity* has been imposed to the *Professional*, which restricts the *Provide Medical Information for Research* goal. As a result of this constraint a *secure goal* is introduced to the *Professional*, *Provide Only anonymous Info*.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have presented extensions to the early requirements stage of the Tropos methodology. Security related concepts and notations were introduced to the existing Tropos concepts in order to allow the modelling of security concerns during the early requirements stage. In addition, to provide formalism for our newly introduced concepts, we have extended Formal Tropos, a specification language amenable to formal analysis.

During the process of extending *Tropos* it was concluded that *Tropos* methodology facilitates the consideration of security requirements for different reasons:

- By considering the overall software development process it is easy to identify security requirements at the early requirements stage and propagate them until the

implementation stage. This introduces a security-oriented paradigm to the software engineering process.

- *Tropos* allows a hierarchical approach towards security. Security would be defined in different levels of complexity, which will allow the software engineer a better understanding while advancing through the process.
- Iteration allows the re-definition of security requirements in different levels therefore providing a better integration with system functionality.
- Consideration of the organisational environment facilitates the understanding of the security needs in terms of the security policy.

Functional and non-functional requirements are defined together however a clear distinction is provided.

As mentioned above the proposed extensions apply only on the early requirements stage of the methodology. However, our aim is to provide a clear well guided process of integrating security and functional requirements throughout the whole range of the development stages. Such a process must use the same concepts and notations throughout the development phases.

Thus, future work involves the assignment of capabilities to the system to help towards the satisfaction of the *secure entities*, and verify the security of the system by analysing potential attacks and if necessary introduce extra secure capabilities. Then, the design of the system will take place by taking into consideration the security analysis performed in the previous stages.

In addition, we are constantly refining and checking the identified concepts, notations, and process by applying them to different real life examples in order to justify them.

REFERENCES

- Castro, J., Kolp, M. and Mylopoulos, J., 2001. A Requirements-Driven Development Methodology. *In Proc. of the 13th Int. Conf. On Advanced Information Systems Engineering (CAiSE'01)*, Interlaken, Switzerland.
- Dardenne, A., Van Lamsweerde, A., Fickas, S., 1993. Goal-directed Requirements Acquisition, *Science of Computer Programming*, 20, pp 3-50.
- Devanbu, P., Stubblebine, S., 2000. Software Engineering for Security: a Roadmap, *Proceedings of the conference of the future of Software Engineering*.

