Using Trust Assumptions in Security Requirements Engineering

Charles B. Haley, Robin C. Laney, Bashar Nuseibeh
Security Requirements Group
Department of Computing
The Open University, UK
{C.B.Haley, R.C.Laney, B.A.Nuseibeh}@open.ac.uk

Jonathan D. Moffett
Department of Computer Science
University of York, UK
jdm@cs.york.ac.uk

Abstract

Assumptions about the trustworthiness of the various components of a system (including human components) can have a significant effect on the specifications derived from the system's requirements. This position paper presents some early efforts to understand the relationships between general requirements, security requirements, and trust assumptions made during problem analysis. An outline of an approach for reasoning about security requirements and trust assumptions is provided.

1. Introduction

Security requirements describe constraints on the functionality of a system under consideration [7]. A requirements engineer should reason about these constraints in the absence of a particular implementation of the system; requirements are *optative*, describing desired behavior instead of existing behavior [5]. Descriptions of the *desired behavior* of individual parts of the system, hereafter referred to as *domains*, are also optative. Descriptions of the *actual behavior* of domains (their inputs, outputs, and states visible at their interfaces) are *indicative*; they describe an objective truth about the behavior of the domain. Unfortunately, as in all things human, there are assumptions behind the meaning of *objectively true*. One could say "a door opens when unlatched and pulled", which is true unless someone has glued the door to the doorframe. One job of an analyst is to make decisions about how much to trust the supplied indicative properties of domains that make up the system. These decisions are *trust assumptions*, and they have a fundamental impact on how the system is realized [9].

To explore the impact of trust assumptions on how a system is realized, we must start by looking at *Requirements Engineering*. Requirements engineering is concerned with enumerating the goals for a system under consideration and producing an optative description of the system's desired behavior [6]. To accomplish this feat, it must be true that a system is intended to solve a given *problem* when placed in a given *context*. A problem description frames the problem, helping ensure that the original intentions are not lost. The context provides indicative information about the domains that are part of the problem.

One can reason about the behavior of a system using *goals*. Goals can be general or specific; general goals must be refined to specific goals. Goals are optative, defining *what* the system is to accomplish (the desired behavior of the system). Once sufficiently refined, goals are *operationalized*, or assigned to a domain that can satisfy the goal. An operationalized goal is a requirement.

All problems involve the interaction of domains that exist in the world. Domains exist either physically (e.g. people) or logically (e.g. data). The Problem Frames notation [5] is useful for diagramming the domains involved in a problem and the interconnection between them. For example, assume that goal reduction produces the requirement "open door when the door-open button is pushed." Figure 1 illustrates the domains that could satisfy the requirement; a basic automatic door system with three domains. One domain is the door mechanism domain, capable of opening and shutting the door. A second is the domain requesting that the door be opened; this domain includes both the 'button' to be pushed and the human pushing the button. The third is the *machine*, the domain being designed to fulfill the requirement that the door open when the button is pushed. The oval presents the requirement.

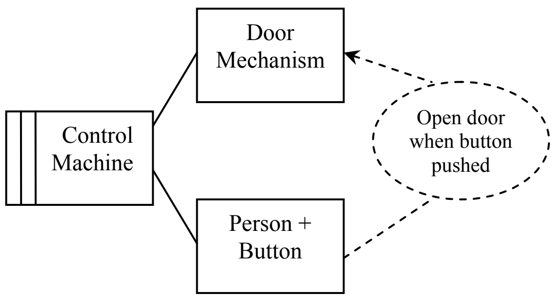

Figure 1 – A basic Problem Frames diagram

Every domain has *interfaces*, which are defined by the *phenomena* visible to other domains. Phenomena are indicative; they can be observed. For example, the person+button domain above might produce the event phenomena ButtonDown and ButtonUp. Alternatively, it might produce the single event OpenDoor. The interplay of phenomena between the machine and the connected domains defines *how* the system accomplishes the goal. The *specification* uses the interplay to describe how the requirements are satisfied [12].

It is very important to note that a specification depends on the phenomena visible at the boundary of the domains, where a requirement does not. For example, in the context of a building we might find the requirements 'permit passage from one room to another' and 'physically separate rooms when possible'. Clearly, these requirements are describing doors. Equally as clearly, they do not specify how the doors operate. The architect must choose the door domain(s) for the system. One might satisfy the requirements with a blanket, a traditional door, a powered automatic door, or some sort of matter transmitter. Each domain implementation presents different phenomena at its boundary (i.e. they work differently), and the resulting system specification must consider these differences. However, the requirements do not change.

When enumerating requirements, it is not sufficient merely to describe a system's behavior. In addition, one must note any security constraints on the behavior. Constraints arising from security concerns are called *security requirements*. The word 'requirement' is important; the security requirements do not specify *how* the constraint is satisfied, only *what* the constraint is.

Continuing with the (admittedly trivial) door example, we might add the security constraint *only family members may pass through an internal door*. How do we know if an individual is a family member? At this point in the process, we do not know. We do know that some domain in the system must be capable of answering the question, and that the specification of that domain must include appropriate phenomena to make the answer known to other domains that need it.

One can easily imagine ways to answer the question. Perhaps a biometric approach is appropriate. Perhaps the matter transmitter will do a DNA test during transmission and reintegrate the person on the lawn if not a family member. Perhaps the only people in the building are family members because some other mechanism (perhaps a security guard) already blocked the interlopers' access.

In the latter case, the answer to the question "is the person a family member?" is known in advance to be 'yes'. However, accepting that proposition requires the analyst to *trust* that some other domain will prevent intruders from entering the building. The other domain claims to satisfy the constraint or participate in its satisfaction. In the example, the analyst *trusts* the other mechanism to satisfy the 'verify family membership' constraint imposed on the machine charged with opening doors. This *trust assumption* fundamentally affects how a requirement is satisfied.

2. Security Requirements

Security requirements express constraints on the behavior of a system. The constraints are intended to limit system behavior to the smallest number of cases possible while still behaving correctly. For example, a goal for an ATM might be 'provide cash to customers'. This goal is obviously overly broad from a security point of view. By providing constraints (security requirements), the circumstances under which cash is provided are reduced. Some examples of constraints might be:

- Only bank customers can get cash
- Only checking account holders can get cash
- The debited account must belong to the customer.

More formally, security requirements concern *operations* on *assets* by *actors*. Without assets, there is nothing of value to protect and constraints are not necessary. If there are no operations (operations change the state of the system-in-the-large in some way), then the system is stable. If there are no actors, then operations are not possible. If any of the three are eliminated, then a discussion about security is uninteresting.

The security community describes the general categories for constraints using the acronym CIA, and more recently another A [8]:

- Confidentiality: limit visibility of the asset to those actors authorized to see it. This is larger than 'read access to a file'. It can include visibility of a data stream on a network or visibility of a paper on someone's desk.
- Integrity: ensure that the asset is not corrupted. As above, this is larger than 'write access to a file', including triggering transactions that should not occur, changing contents of backup media, or making incorrect entries in a paper-based accounting system.
- Accessibility: ensure that the asset is readily accessible to actors that need it. The common counterexample is preventing a company from doing business by denying it access to something important, such as access to its computer systems or its offices.
- Authentication: ensure that the provenance of the asset or actor is known. A common example is the simple login. More complicated examples include mutual authentication (e.g. exchange of cryptography keys), and intellectual property rights management.

One reasons about security constraints by applying a qualitative reasoning method using these categories.

3. Trust Assumptions

Recall that a requirement describes what a system is to accomplish. From the point of view of the analyst, how a requirement is satisfied depends on the characteristics of the domains in the problem. An analogous relationship exists between security requirements and trust assumptions; how security requirements are satisfied depends on the trust assumptions made by the analyst.

For our purposes, we use the definition of trust proposed by Grandison & Sloman [3]: "[Trust] is the quantified belief by a trustor with respect to the competence, honesty, security and dependability of a trustee within a specified context". In our case, the *analyst* trusts that some domain will participate 'competently and honestly' in the satisfaction of a constraint.

The notion of trust assumptions can be illustrated using the door example. The constraint 'only family members may pass through an internal door' was attached to the requirement. The analyst's trust assumptions could lead the analyst to propose the following specifications:

- Assumption 1: only trust the problem context controlled by the analyst. A domain must exist in the context that can verify family membership for an internal door, and a door controller must exist that is able to make use of the check. For example, the family-member check domain might perform a DNA test or a voiceprint test, and then pass the results to the door controller. Alternatively, a guard can be posted at each door, and the guard decides who gets through by pushing the button to open the door.
- Assumption 2: the outer door to the building, which is not in the context of the problem, is trusted. All people passing through that door are verified as family members, thus all people passing through the inner door are family members. Not only do we not need a family-member-checking domain, but also we might not need a door controller domain. A blanket will do for a door.
- Assumption 3: only one family lives on the planet. All people entering the building can only be from one family, satisfying the requirement. No check at the outer door of the building is needed. This example shows that the analyst can trust a domain extremely far removed from the immediate problem.

The cases show that the domains that are needed to realize the system depend on the analyst's trust assumptions. In the first case, the system contains a family-membership checker and a door controller, and the two must interact to function properly. In the second and third cases, no family-membership-check domain is needed. The same requirement applies to all three cases, but the realizations of the requirement are radically different.

Consider the ATM example presented earlier. A bank may tell the analyst that it trusts its customers to be honest, but that it trusts no one else. Using this trust assumption, the analyst could make several architectural decisions. The ATM must somehow verify that the person is an account holder. It might then ask the person for the account number and the amount, and simply supply the cash. Alternatively, the ATM might be a special booth that customers can enter after being authenticated via a retinal scan or by a security guard. The customer finds a bag of money and a pad of paper in the booth, takes as much money as desired, writes the amount and the account number on the paper, and leaves. Both systems conform to the requirements, given the trust assumptions the analyst makes.

More formally, a trust assumption is an assumption by an analyst that the specification of a domain can depend on certain properties of some other domain in order to satisfy a security requirement. The analyst *trusts* the *assumption* to be true. The assumption results in a relation between two domains; the satisfaction of a security requirement by one domain depends upon the properties of another. This dependency relation is represented as a directed arc from the depending domain to the domain considered to have the necessary properties.

To ensure that the specification conforms to the security requirements, every security constraint must be matched with some dependency relation(s). By creating the relations, the analyst is asking the 'owners' of the depended-upon domains to agree to the 'contract' implied by the relation. The target domains in the dependency relations are not necessarily in the context of the problem being solved. Figure 2 illustrates this by overlaying the dependency relation onto the diagram from Figure 1, using the second assumption: that there is a guard at the front door of the building.

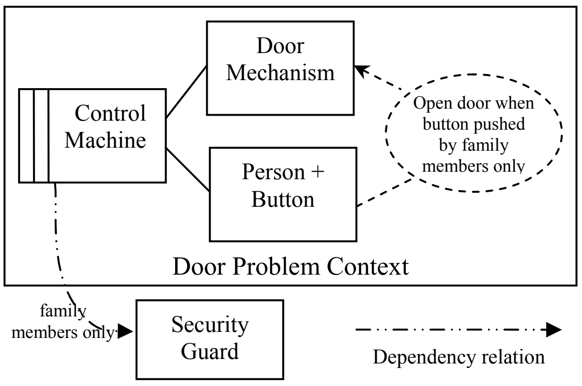

Figure 2 – Problem diagram showing the dependency

The dependency relations document the analyst's trust assumptions. The analyst verifies the set of assumptions described by the relations, satisfying him- or herself that the assumptions are valid and consistent. The relations are also useful during the security validation process, during which the depended-upon properties must be verified.

4. Related Work

Several teams are looking at the role of trust in security requirements engineering. In the *i** framework [10, 11], Yu, Lin, & Mylopoulos take an 'actor, intention, goal' approach where security and trust relationships within the model are modeled as “softgoals”: goals that have no quantitative measure for satisfaction. The Tropos project [2] uses the *i** framework, adding on wider lifecycle coverage. Gans et al [1] add distrust and “speech acts”. None of the models capture the analyst’s assumptions about the domains that make up the solution to the problem. As such, an *i** model complements the approach presented here, and in fact can be used to determine the goals and requirements.

He and Antón [4] are concentrating on privacy, working on mechanisms to assist people to trust privacy policies, for example on web sites. They propose a context-based access model. Context is determined using “purpose” (why is information being accessed), “conditions” (what conditions must be satisfied before access can be granted), and “obligations” (what actions must be taken before access can be granted. The framework describes run-time properties, not the analyst’s assumptions about the domains forming the solution.

5. Conclusions and Future Work

We have provided an informal approach for reasoning about security requirements. The approach makes a strong distinction between system requirements and machine specifications, permitting the analyst to choose how to conform to the requirements. The trust assumptions embedded in the domain inform the analyst, enabling the analyst to choose between alternate ways of satisfying the requirements.

There is a great deal of work left to do. Our approach starts with requirements derived from goals before reasoning about security constraints, so integrating the approach with a goal-oriented method would help. KAOS ([6] etc) is a good candidate as may be *i**. Such an integration would profit greatly from having a *logic* to express the security requirements; analysis could then to some extent be automated.

The qualitative reasoning approach would be helped by guidance for determining security requirements. The security categories assist by focusing on crosscutting concerns, but these need further elaboration. Templates could help the process, as could taxonomies of standard approaches.

Capturing trust assumptions in some semi-formal way would help ensure that the resulting relations are consistent; the logic developed in the SULTAN project holds promise [3]. It also seems that there are *layers* of assumptions, where one assumption is in reality part of a larger one; representing these layers would be a step forward. Lastly, incorporating *risk* into the dependency relations would help resolve problems exemplified by statements of the form “I trust you, but …”

Acknowledgements: The financial support of the Leverhulme Trust is gratefully acknowledged. Thanks also go to Michael Jackson for many insights about Problem Frames and requirements, and to Anthony Meehan for his comments that started us down this path.

References

1. Gans, G., et al. "Requirements Modeling for Organization Networks: A (Dis)Trust-Based Approach," In *5th IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada: IEEE Computer Society Press, 27-31 Aug 2001, 154-165.

2. Giorgini, P., Massacci, F., & Mylopoulos, J. *Requirement Engineering Meets Security: A Case Study on Modelling Secure Electronic Transactions by VISA and Mastercard*, Department of Information and Communication Technology, DIT-03-027. University of Trento, May 2003.

3. Grandison, T., & Sloman, M. "Trust Management Tools for Internet Applications," In *The First International Conference on Trust Management*. Heraklion, Crete, Greece: Springer Verlag, 28-30 May 2003.

4. He, Q., & Antón, A. I. "A Framework for Modeling Privacy Requirements in Role Engineering" in Ninth International Workshop on Requirements Engineering: Foundation for Software Quality, *The 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*, Klagenfurt/Velden, Austria, 16 Jun 2003.

5. Jackson, M. *Problem Frames*. Addison Wesley, 2001.

6. van Lamsweerde, A. "Goal-Oriented Requirements Engineering: A Guided Tour," In *5th IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada: IEEE Computer Society Press, 27-31 Aug 2001, 249-263.

7. Moffett, J. D., & Nuseibeh, B. *A Framework for Security Requirements Engineering*, Department of Computer Science, YCS368. University of York, UK, 2003.

8. Pfleeger, C. P., & Pfleeger, S. L. *Security in Computing*. Prentice Hall, 2002.

9. Viega, J., Kohno, T., & Potter, B. "Trust (and Mistrust) in Secure Applications," *Communications of the ACM* 44(2), Feb, 2001: 31-36.

10. Yu, E. "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering," In *Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE'97)*. Annapolis MD USA, 6-10 Jan 1997, 226-235.

11. Yu, E., & Liu, L. "Modelling Trust for System Design Using the i* Strategic Actors Framework," In R. Falcone, M. P. Singh, & Y.-H. Tan, eds. *Trust in Cyber-societies, Integrating the Human and Artificial Perspectives*. Springer-Verlag Heidelberg, 2001: 175-194.

12. Zave, P., & Jackson, M. "Four Dark Corners of Requirements Engineering," *ACM Transactions on Software Engineering and Methodology* 6(1), Jan, 1997: 1-30.