Elaborating Security Requirements by Construction of Intentional Anti-Models

Axel van Lamsweerde
Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

Abstract

Caring for security at requirements engineering time is a message that has finally received some attention recently. However, it is not yet very clear how to achieve this systematically through the various stages of the requirements engineering process.

The paper presents a constructive approach to the modeling, specification and analysis of application-specific security requirements. The method is based on a goal-oriented framework for generating and resolving obstacles to goal satisfaction. The extended framework addresses malicious obstacles (called anti-goals) set up by attackers to threaten security goals. Threat trees are built systematically through anti-goal refinement until leaf nodes are derived that are either software vulnerabilities observable by the attacker or anti-requirements implementable by this attacker. New security requirements are then obtained as countermeasures by application of threat resolution operators to the specification of the anti-requirements and vulnerabilities revealed by the analysis. The paper also introduces formal epistemic specification constructs and patterns that may be used to support a formal derivation and analysis process. The method is illustrated on a web-based banking system for which subtle attacks have been reported recently.

1. Introduction

Security has become an increasingly growing concern in the internet age. The number of security incidents reported has been growing exponentially over the past decade [7]. Software applications are increasingly ubiquitous, heterogeneous, mission-critical and vulnerable [19]. Attackers are more and more malicious and use increasingly sophisticated attack technology. The consequences of attacks may become more devastating up to the point of breaking severe safety-critical concerns. For example, there have been reports about denial of service on medical records that prevented urgent surgery from being undertaken under the right conditions [7]. Unsurprisingly, the major source of vulnerability has been recognized to be poor-quality software [41].

The state of the art in security engineering has been fairly unbalanced so far. As Wing pointed out, the "strength" of security guarantee provided by current security technology is inversely proportional to the "size" of the software layer at which the technology applies [42]. At the bottom, the *crypto layer* offers solid and well-established techniques for basic services such as encryption/decryption or signature; the state of the art on this layer precisely tells us what can be guaranteed, what cannot and what are the problems still left open [37]. Above the crypto layer, the *security protocol* layer offers a wide range of standard procedures for services such as secure communication, authentication or key exchange; the state of the art on this layer provides us with specific logics [5] and formal techniques for verifying security protocols to point out errors or hidden assumptions [18, 29, 13, 9]. Above the security layer, the *system* layer provides standard services, implemented in some programming *language,* such as remote file access; services like SSH, SSL or SSHTP and language technologies like Authenticode, Active X or Java provide some level of security but are subject to multiple types of attacks such as denial of service or spoofing. Above the system layer, the *application* layer offers services such as web-based banking operations that must implement application-specific security requirements in terms of primitives from lower layers. The state of the art in security enginering at the application layer is much more limited [42, 41].

This paper focusses on security enginering at the application layer exclusively. A necessary condition for application software to be secure is obviously that all application-specific security requirements be met by the software. Such requirements must therefore be engineered with great care. They need to be explicit, precise, adequate, non-conflicting with other requirements and *complete*. In particular, application-specific requirements should anticipate application-specific attack scenarios such as, e.g., attacks on a web-based banking application that may result in disclosure of sensitive information about bank accounts or in fraudulous money transfer. Security requirements engineering must thus address a broader system perspective where the software *environment* is explicitly modeled and analyzed as well.

The elaboration, specification, analysis and documentation of application-specific security requirements is an area that has been left virtually unexplored by requirements engineering research to date (with a very few recent exceptions discussed at the end of this paper). A requirements engineering (RE) technique for security-critical systems should ideally meet the following meta-requirements.

- *Early deployment:* In view of the criticality of security requirements, the technique should be applicable as early as possible in the RE process, that is, to declarative assertions as they arise from stakeholder interviews and documents (as opposed to, e.g., later state machine models).
- *Incrementality:* The technique should support the intertwining of model building and analysis and therefore allow for reasoning about partial models.
- *Reasoning about alternatives:* The technique should make it possible to represent and assess alternative options so that a "best" route to security can be selected.
- *High assurance:* The technique should allow for formal analysis *when* and *where* needed so that compelling evidence of security assurance can be provided.
- *Security-by-construction:* To avoid the endless cycle of defect fixes generating new defects, the RE process should be guided so that a satisfactory level of security is guaranteed by construction.
- *Separation of concerns:* the technique should keep security requirements separate from other types of requirements so as to allow for interaction analysis [20, 35].

This paper presents a method for elaborating security requirements aimed at addressing the above meta-requirements. The general idea is to build two models iteratively and concurrently:

- a model of the system-to-be that covers both the software and its environment and inter-relates their goals, agents, objects, operations, requirements and assumptions;
- an anti-model, derived from the model, that exhibits how specifications of model elements could be maliciously threatened, why and by whom.

In this overall picture, security requirements are elaborated systematically by iterating the following steps: (a) instantiate specification patterns associated with property classes such as confidentiality, privacy, integrity, availability, authentication or non-repudiation; (b) derive anti-model specifications threatening such specifications; (c) derive alternative countermeasures to such threats and define new requirements by selection of alternatives that best meet other quality requirements from the model.

Our method builds on a goal-oriented framework we developed before for generating and resolving obstacles to requirements achievement [21, 22]. Our extension to malicious obstacles allows the obstacle refinement process to be guided by the attacker's own goals and by the target of deriving observable vulnerabilities and implementable threats. The extension also includes security-specific operators for resolving malicious obstacles.

The paper is organized as follows. Section 2 briefly reviews some background material on goal-oriented requirements engineering used in the sequel. Section 3 introduces security specification patterns together with some epistemic extensions to our real-time temporal logic needed to formalize them. Section 4 shows how our techniques for generating and resolving obstacles to goal achievement are extended to integrate malicious obstacles set up by attackers who want to break security goals. Section 5 addresses the generation of countermeasures. Section 6 then illustrates our method on the engineering of security requirements for web-based banking services; we show in particular how a critical episode from a real attack recently reported can be generated formally using our technique. To conclude, we summarize the contribution, compare it with related efforts and discuss various issues left open by our approach.

2. Background

A *goal* is a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some of the agents forming that system. *Agents* are active components such as humans, devices, legacy software or software-to-be components that play some *role* towards goal satisfaction. Some agents thus define the software whereas others define the environment. Goals may refer to services to be provided (functional goals) or to quality of service (non-functional goals). Unlike goals, *domain properties* are descriptive statements about the environment such as physical laws, organizational norms or policies, etc.

Goals are organized in AND/OR *refinement-abstraction hierarchies* where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve less agents [11, 12]. In such structures, *AND-refinement* links relate a goal to a set of subgoals (called *refinement*) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links may relate a goal to a set of alternative refinements; this means that satisfying one of the refinements is a sufficient condition in the domain for satisfying the goal.

Goal refinement ends when every subgoal is *realizable* by some individual agent assigned to it, that is, expressible in terms of conditions that are monitorable and controllable by the agent [24]. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment (unlike requirements, expectations cannot be enforced by the software-to-be).

Goals prescribe intended behaviors; they can be formalized in a real-time temporal logic [11]. Keywords such as Achieve, Avoid, Maintain are used as a lightweight alternative to characterize goals according to the temporal behavior pattern they prescribe. *Softgoals* prescribe preferred behaviors; they can be used for selecting preferred alternatives in an AND/OR goal refinement graph [8].

Goals are *operationalized* into specifications of operations to achieve them [11, 25]. In the specifcation of an operation, a distinction is made between *domain* pre- and postconditions that capture what any application of the operation means in the application domain, and *required* pre-, trigger, and postconditions that capture requirements on the operations that are necessary for achieving the underlying goals.

Goals refer to *objects* that can be incrementally derived from their specification to produce a structural model of the system (represented by UML class diagrams). Objects have states defined by their attributes and links to other objects; they are passive (entities, associations, events) or active (agents). Agents are related together via their interface made of object attributes they *monitor* and *control*, respectively [32].

Obstacles were first introduced in [33] as a means for identifying goal violation scenarios. In declarative terms, an *obstacle* to some goal is a condition whose satisfaction may prevent the goal from being achieved. An obstacle *O* is said to *obstruct* a goal *G* in some domain characterized by a set of domain properties *Dom* iff

$\{O, Dom\} \models \neg G$ *obstruction*

$Dom \not\models \neg O$ *domain consistency*

Obstacle analysis consists in taking a pessimistic view at the goals, requirements and expectations being elaborated. The principle is to identify as many ways of obstructing them as possible in order to resolve each such obstruction when likely and critical so as to produce more complete requirements for more robust systems. Formal techniques for generation and AND/OR refinement of obstacles are available [22]. The basic technique amounts to a precondition calculus that regresses goal negations *ØG* backwards through known domain properties *Dom*. Formal obstruction patterns may be used as a cheaper alternative to shorcut formal derivations. Both techniques allow domain properties involved in obstructions to be incrementally elicited as well.

Obstacles that appear to be likely and critical need to be resolved once they have been generated. Resolution tactics are available for generating alternative resolutions, notably, *goal substitution*, *agent substitution*, *goal weakening*, *goal restoration*, *obstacle prevention* and *obstacle mitigation* [22]. The selection of preferred alternatives depends on the degree of criticality of the obstacle, its likelihood of occurrence and on high-priority softgoals that may drive the selection. The selected resolution may then be deployed at specification time, resulting in specification transformation [22], or at runtime through obstacle monitoring [15].

In the context of this paper,

- security concerns are captured by security goals that need to be made precise and refined until reaching *security requirements* on the software and expectations on the environment (the latter may capture *security policies* in the environment);
- *attackers* are malicious agents in the environment;
- *threats* are obstacles intentionally set up by attackers;
- *assets* to be protected against threats correspond to passive or active objects.

There is thus no need for introducing additional abstractions in the underlying meta-model to cope with security; our RE environment kernel can therefore be used for graphical editing, model querying, report generation, traceability management, goal model checking and goal-oriented animation [31, 34].

3. Specification Patterns for Security Goals

The preliminary elicitation of security-related goals is driven by application-specific instantiations of generic specification patterns. The patterns are associated with specializations of the SecurityGoal meta-class, namely, *Confidentiality*, *Integrity*, *Availability, Privacy, Authentication* and *Non-repudiation* goal subclasses. Patterns refer to meta-classes from the language meta-model (such as Object and Agent). For each subclass of SecurityGoal, the instantiation of the corresponding specification pattern to "sensitive" attributes/associations from the object model yields corresponding candidates for application-specific security goals (the latter may then need to be refined if necessary).

To support formal analysis, our specification patterns may be formalized in a first-order, real-time linear temporal logic [22] augmented with epistemic constructs for security-related predicates.

In particular, the epistemic operator $KnowsV_{ag}$ is defined on state variables as follows:

$KnowsV_{ag}(v) \equiv \exists x: Knows_{ag}(x = v)$ ("knows value")

$Knows_{ag}(P) \equiv Belief_{ag}(P) \land P$ ("knows property")

where the operational semantics of the epistemic operator $Belief_{ag}(P)$ is "*P is among the properties stored in the local memory of agent ag*". Domain-specific axioms must make it precise under which conditions property *P* does appear and disappear in the agent's memory. An agent thus *knows a property* if that property is found in its local memory and it is indeed the case that the property holds.

The following temporal logic notations are used in this paper: "$P \Rightarrow Q$" means "$\Box (P \rightarrow Q)$" where the temporal operator "$\Box$" means "in every future state" and "$\rightarrow$" denotes logical implication; "**o**" means "in the next state"; "$\Diamond_{\leq d}$" means "some time in the future within *d* time units".

For example, the specification pattern for Confidentiality goals defines confidentiality in a generic way:

Goal *Avoid* [SensitiveInfoKnownByUnauthorizedAgent]
FormalSpec $\forall$ ag: Agent, ob: Object
$\neg$ Authorized (ag, ob.Info) $\Rightarrow$ $\neg$ $KnowsV_{ag}$ (ob.Info)

The Authorized predicate is generic and has to be instantiated through an application-specific *domain definition*. For example, for web-based banking we would certainly consider the instantiation Object/Account while searching through the object model for sensitive information; we might then introduce the following instantiating definition:

$\forall$ ag: Agent, acc: Account
Authorized (ag, acc) $\equiv$
Owner (ag, acc) $\lor$ Proxy (ag, acc) $\lor$ Manager (ag, acc)

Sensitive information about accounts includes the pair of objects (Acc#, PIN). The latter are defined in the object model as partOf the aggregation Account and interrelated through the association *Matching*.

The instantiation of the *Confidentiality* specification pattern to such sensitive information yields the following confidentiality goal as candidate for inclusion in the goal model:

Goal *Avoid* [AccountNumber&PinKnownByUnauthorized]
FormalSpec $\forall$ p: Person, acc: Account
$\neg$ (Owner (p, acc) $\lor$ Proxy (p, acc) $\lor$ Manager (p, acc))
$\Rightarrow$ $\neg$ [$KnowsV_p$ (acc.Acc#) $\land$ $KnowsV_p$ (acc.PIN)])

The same principle may be used for eliciting instantiations of the following specification patterns for Privacy, Integrity and Availability goals:

Goal *Maintain* [PrivateInfoKnownOnlyIfAuthorizedByOwner]
FormalSpec $\forall$ ag, ag': Agent, ob: Object
$KnowsV_{ag}$ (ob.Info) $\land$ OwnedBy (ob.Info, ag') $\land$ ag $\neq$ ag'
$\Rightarrow$ AuthorizedBy (ag, ob.Info, ag')

Goal *Maintain* [ObjectInfoChangeOnlyIfCorrectAndAuthorized]
FormalSpec $\forall$ ag: Agent, ob: Object, v : Value
ob.Info = v $\land$ **o** (ob.Info $\neq$ v) $\land$ UnderControl (ob.Info, ag)
$\Rightarrow$ Authorized (ag, ob.Info) $\land$ **o** Integrity (ob.Info)

Goal *Achieve* [ObjectInfoUsableWhenNeededAndAuthorized]
FormalSpec $\forall$ ag: Agent, ob: Object, v : Value
Needs (ag, ob.Info) $\land$ Authorized (ag, ob.Info)
$\Rightarrow$ $\Diamond_{\leq d}$ Using (ag, ob.Info)

Specifications of application-specific security goal candidates are thus obtained from such specification patterns by (a) instantiating meta-classes such as Object, Agent and generic attributes such as *Info* to application-specific sensitive classes, attributes and associations in the object model, and (b) specializing predicates such as Authorized, UnderControl, Integrity or Using through substitution by application-specific definitions.

In the context of security it may be worth recalling that sensitive objects found in the object model may be either passive (entities, associations, events) or active (agents, e.g., programs).

4. Building Intentional Threat Models

As noted in [22], obstacles may obstruct safety or security goals; obstacle refinement trees then correspond to the popular *fault trees* used for modeling or documenting hazards in safety-critical systems [26] and to the popular *threat trees* used for modeling or documenting potential attacks in security-critical systems [2, 38, 30, 17]. There are two significant differences, however.

- Obstacle trees are *goal-anchored* as their root is a goal negation; the analyst thus knows exactly where to start the analysis from - it is often much easier to concentrate first on what is desired rather than on what is not desired.
- Obstacles can be formally generated by regressing goal negations through domain properties and other goal assertions, or by using formal obstruction and refinement patterns [20, 22, 12].

In the context of security engineering, standard obstacle analysis appears to be too limited for handling malicious obstacles; the reasons are the following.

- The goals underlying malicious obstacles are not captured; one can therefore not use them for driving the obstacle refinement process.

- There is no modeling of attacker agents and their capabilities in terms of operations they can perform and objects they can monitor/control; one can therefore not reason about them.
- Software vulnerabilities are not explicit in standard models; one can therefore not reason about them or, better, derive them.
- The outcome of the obstacle likelihood and criticality assessment process may be quite different; for example, standard obstacle analysis for a ground collision avoidance component of an air traffic control system might miss the obstacle of multiple planes crashing into adjacent buildings at almost the same time, or assess it to be extremely unlikely, whereas the explicit incorporation of terrorist agents and their goal of causing major damage to symbols of economic power might result in totally different conclusions.

Richer models should thus be built to capture attackers, their goals and capabilities, the software vulnerabilities they can monitor or control, and attacks that satisfy their goals based on their capabilities and on the system's vulnerabilities.

Let us call *anti-models* such models and *anti-goals* the attacker's own goals, including malicious obstacles to security goals. Anti-goals should of course be distinguished from the goals the system under consideration should satisfy. Anti-models should lead to the generation of more subtle threats and the derivation of more robust security requirements as anticipated countermeasures to such threats.

Table 1 describes a procedure for building intentional anti-models in a systematic way. This procedure corresponds to a *dual* version of the goal-oriented requirements elaboration method we had extensive experience with for many years [23, 31]. The steps there are ordered by data dependencies and generally intertwined.

1. Get initial anti-goals by negating relevant Confidentiality, Privacy, Integrity and Availability goal specification patterns instantiated to sensitive objects from the object model.
2. For each such anti-goal, elicit potential attacker agents that might own the anti-goal, from questions such as "*WHO can benefit from this anti-goal?*" (Application-specific specializations of known attacker taxonomies may help answering such questions).
3. For each anti-goal *and* corresponding attacker class(es) identified, elicit the attacker's higher-level anti-goals from questions such as "*WHY would instances of this attacker class want to achieve this anti-goal?*". Such questions may be asked recursively to elicit more and more abstract anti-goals yielding threat rationales together with other potential threats from alternative refinements of those higher-level anti-goals.
4. Elaborate the anti-goal AND/OR graph by AND-refining/abstracting anti-goals along alternative branches, with the aim of deriving terminal anti-goals that are realizable either by the identified attacker agents or by attackee software agents. The former are *anti-requirements* assigned to the attacker whereas the latter are *vulnerabilities* assigned to the attackee. (This step may be performed informally by asking HOW/WHY questions [23], or formally by regression through the goal model and the domain theory [20] or by use of refinement patterns [12, 24] and obstruction patterns [22].)
5. Derive the object and agent anti-models from anti-goal specifications. The boundary between the anti-machine (under the attacker's control) and the anti-environment (which includes the software attackee) are thereby derived together with monitoring/control interfaces [24].
6. AND/OR-operationalize all anti-requirements in terms of potential capabilities of the corresponding attacker agent [25] – the latter may include blind or intelligent searching, eavesdropping, deciphering, spoofing, cookie installation, etc.

Table 1: Anti-Model Building Method

In Step 4, an anti-goal is said to be *realizable* by some agent if it is formulated in terms of conditions monitorable and controllable by the agent. (A more technical definition of realizability may be found in [24].) Note that the vulnerabilities derived by anti-goal refinement in Step 4 rightly fit the Common Criteria definition of *vulnerability*, namely, "a condition of an agent that, in conjunction with a threat, can lead to security requirement violation" [6].

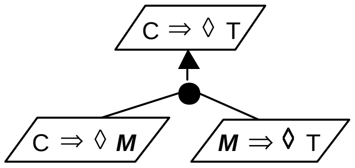
As indicated in Step 4, the AND/OR refinement/abstraction process may be guided by the following techniques:

- asking "HOW?" and "WHY?" questions about the anti-goals already found;
- regressing anti-goal specifications through domain properties to find out anti-goal preconditions that are satisfiable in the domain [22] – this corresponds to situations where the attacker exploits features of the domain to achieve her anti-goals;
- regressing anti-goal specifications through goal specifications from the primal model to find out anti-goal preconditions that are satisfiable by the software [20] – this corresponds to situations where the attacker exploits features of the software itself to achieve her

anti-goals;

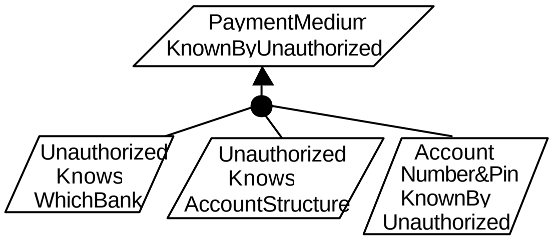
- applying formal refinement patterns – notably, the "milestone", "decomposition-by-case", "resolve lack of monitorability" and "resolve lack of controllability" patterns [12, 24] and obstruction patterns [22].

For example, the "milestone" pattern in Fig.1 may lead for a web banking application to the refinement of the anti-goal PaymentMediumKnownByUnauthorized shown in Fig.2 (the bold circle associated with the AND-node there indicates that the refinement is provably complete).



$C \Rightarrow \Diamond T$
$C \Rightarrow \Diamond \boldsymbol{M}$
$\boldsymbol{M} \Rightarrow \Diamond T$

Fig. 1 – The milestone pattern



PaymentMedium KnownByUnauthorized
Unauthorized Knows WhichBank
Unauthorized Knows AccountStructure
Account Number&Pin KnownBy Unauthorized

Fig. 2 – Milestone pattern application

Section 6 illustrates the use of the anti-model building method in greater detail; in particular, it shows how anti-goals are regressed formally through domain properties. The resulting anti-model relates attackers, their anti-goals, referenced objects and anti-goal operationalizations to the attackees, their goals, objects, operations and vulnerabilities.

When goals are formalized in our real-time temporal logic, the logical models of anti-goals are sets of *attack scenarios*. Bounded SAT solvers can then be used to generate such scenarios automatically [34].

5. Generating Countermeasures

Once intentional anti-models have been built systematically in this way, the next step in the model/anti-model building cycle is to consider alternative countermeasures to the various vulnerabilities and anti-requirements found. A preferred countermeasure is then selected based on (a) the severity and likelihood of the corresponding threat, and (b) non-functional goals that have been identified in the primal goal model. The selected countermeasure yields a new security goal to be integrated in the latter model.

Alternative countermeasures may be produced systematically using operators similar to those described in [22] for resolving obstacles, e.g.,

- *Goal substitution*: develop an alternative refinement of the parent goal to prevent the original subgoal from being threatened by the anti-goal;
- *Agent substitution*: replace a vulnerable agent assigned to a threatened goal by a less vulnerable one for the threatening anti-goal;
- *Goal weakening*: weaken the specification of the goal being threatened so as to make it circumvent the threatening anti-goal;
- *Goal restoration*: introduce a new goal prescribing appropriate restoration measures in states where the goal has been threatened by the anti-goal;
- *Anti-goal mitigation*: tolerate the anti-goal but mitigate its effects;
- *Anti-goal prevention*: add a new goal requiring the anti-goal to be Avoided.

The above list may be extended with *security-specific* operators such as the following:

- *Protect vulnerability*: make the derived vulnerability condition unmonitorable by the attacker.
- *Defuse threat:* make the derived anti-requirement condition uncontrollable by the attacker.
- *Avoid vulnerability:* add a new goal requiring the software vulnerability condition to be Avoided.

Once alternative anti-goal resolutions have been produced by application of such operators a preferred one has to be selected based on how critical the goal being threatened is and on how well the resolution meets other non-functional goals. The NFR qualitative framework may be used here to support the selection process [8].

The new security goal thereby retained has to be AND/OR refined in turn until requirements/expectations are reached. A new model/anti-model building cycle may then be required.

For anti-goals that are not too severe, resolutions may be deferred from specification time to run time using anti-goal monitoring and intrusion detection technology [15, 16, 4].

6. Case Study: Web-Based Banking Services

We apply our anti-model building method for web-based banking services and then illustrate what the resolutions might look like.

Step 1 results in instantiating the confidentiality goal pattern *Avoid* [SensitiveInfoKnownByUnauthorizedAgent] into:

Avoid [AccountNumber&PinKnownByUnauthorized]

(see Section 3). The instantiated formal specification is then negated which yields

AntiGoal *Achieve* [AccountNumber&PinKnownByUnauthorized]
FormalSpec $\lozenge \exists$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ $KnowsV_p$ (acc.Acc#) $\wedge$ $KnowsV_p$ (acc.PIN)

By asking *WHO* could benefit from the above anti-goal one could in Step 2 elicit potential attacker agent classes such as Thief, Hacker, BankQualityAssuranceTeam, etc.

Consider the case of a Thief agent, for example. Step 3 would lead to the elicitation of a parent anti-goal such as

Achieve [PaymentMediumKnownByThief]

and the grand-parent anti-goal

Achieve [MoneyStolenFromBankAccounts]

Lack of space prevents us from building the entire anti-goal, anti-object and anti-operation models associated with the Thief agent through steps 4-6 of the anti-model building method. For example, the milestone pattern instantiation in Fig. 2 produces two other child nodes of the anti-goal Achieve[PaymentMediumKnownByThief], shown in Fig. 3, namely,

Achieve [ThiefKnowsWhichBank] ,
Achieve [ThiefKnowsAccountStructure]

Let us now focus on the derivation of refinements for the antigoal

Achieve [AccountNumber&PinKnownByUnauthorized]

along one branch of the anti-goal AND/OR graph.

Looking at the above formal specification of this anti-goal we ask ourselves "what are sufficient conditions in the domain for someone unauthorized to know both the number and PIN of an account simultaneously?". We may also use the symmetry of the association *Matching* between account numbers and PINs in the object model and its multiplicity [1..1, 1..N]. As a result we elicit two symmetrical *domain properties*, namely,

$\forall$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ $KnowsV_p$ (acc.Acc#)
$\wedge$ ($\exists$ x: PIN) (Found (p, x) $\wedge$ Matching (x, acc.Acc#)
$\Rightarrow$ $KnowsV_p$ (acc.Acc#) $\wedge$ $KnowsV_p$ (acc.PIN)

and

$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ $KnowsV_p$ (acc.PIN)
$\wedge$ ($\exists$ y: Acc#) (Found (p, y) $\wedge$ Matching (acc.PIN, y)
$\Rightarrow$ $KnowsV_p$ (acc.Acc#) $\wedge$ $KnowsV_p$ (acc.PIN)

We may now formally regress the above anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized] through each of these domain properties to obtain two sub-goals as alternative preconditions for achieving this anti-goal. (The technique amounts to a kind of backward chaining through LTL properties, see [22] for details.) We thereby obtain an OR-refinement of that anti-goal into two alternative, symmetrical anti-subgoals, namely,

AntiGoal *Achieve* [AccountKnown&MatchingPinFound]
FormalSpec $\lozenge \exists$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ $KnowsV_p$ (acc.Acc#)
$\wedge$ ($\exists$ x: PIN) [Found (p, x) $\wedge$ Matching (x, acc.Acc#)]

and

AntiGoal *Achieve* [PinKnown&MatchingAccountFound]
FormalSpec $\lozenge \exists$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc)]
$\wedge$ $KnowsV_p$ (acc.PIN)
$\wedge$ ($\exists$ y: Acc#) [Found (p, y) $\wedge$ Matching (acc.PIN, y)]

The refinement process goes on until software vulnerabilities and anti-requirements realizable by the Thief agent are reached. Fig. 3 shows a portion of the anti-goal graph built thereby.

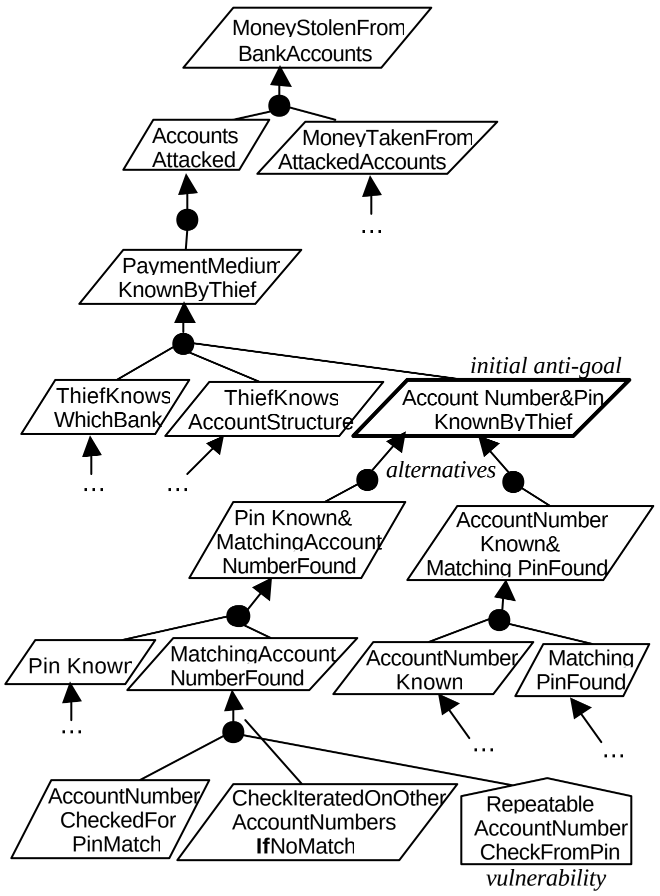

MoneyStolenFrom BankAccounts
Accounts Attacked
MoneyTakenFrom AttackedAccounts
PaymentMedium KnownByThief
initial anti-goal
ThiefKnows WhichBank
ThiefKnows AccountStructure
Account Number&Pin KnownByThief
alternatives
Pin Known& MatchingAccount NumberFound
AccountNumber Known& Matching PinFound
Pin Known
MatchingAccount NumberFound
AccountNumber Known
Matching PinFound
AccountNumber CheckedFor PinMatch
CheckIteratedOnOther AccountNumbers IfNoMatch
Repeatable AccountNumber CheckFromPin
vulnerability

Fig. 3 – Portion of an anti-goal model for web banking services

The derived anti-requirements assignable to the Thief agent are, in one alternative refinement,

PinCheckedForAccountNumberMatch,
CheckIteratedOnOtherPins**If**NoMatch

and, in the other alternative,

AccountNumberCheckedForPinMatch,
CheckIteratedOnOtherAccountNumbers**If**NoMatch

Those anti-requirements are operationalizable using web technology, see [36] for details. The anti-requirements in the second alternative is more subtle and corresponds to a sophisticated, real attack reported in [36].

The corresponding vulnerabilities we derived are

RepeatablePinCheckFromAccountNumber

and

RepeatableAccountNumberCheckFromPin,

respectively.

Once this anti-model has been built we need to move to countermeasures resulting in new security goals. In this case the resolution operator

Avoid [Vulnerability]

seems required in view of the criticality of the vulnerabilities. This will produce two new, *conjoined* security goals:

Avoid [RepeatablePinCheckFromAccountNumber],
Avoid [RepeatableAccountNumberCheckFromPin]

The first new goal corresponds to the usual limitation of the number of PIN entries (e.g., to three attempts). The second new goal prevents the possibility of exhaustive searching for account numbers that match some fixed PIN number. These goals have to be refined in turn until requirements on the software and expectations on the environment are reached. A new anti-model building cycle may be undertaken for these new goals, if necessary.

7. Related work

A few proposals have been made recently for extending existing modeling notations to capture attacker features at requirements engineering time - notably, through misuse cases that complement UML use cases [40, 1]. Anti-requirements were suggested as requirements on the attacker to break existing requirements [10]. This notion is elaborated further in [27] where abuse frames are introduced to define the scope and boundary of anti-requirements.

Goal-based approaches have already been used to address security concerns at requirements engineering time, notably, through security goal taxonomies. For example, the NFR framework allows known security properties to be organized as AND/OR goal graphs [8]. Anton, Earp and Reese propose a rich taxonomy of privacy goals based on their analysis of 23 internet privacy policies for a variety of companies in health care industries [3].

The principle of building a catalogue of known threat tree patterns for documentation and reuse through instantiation is nicely illustrated in [30].

The work closest in spirit to ours is [28]; they propose an extension of the *i** agent-based requirements engineering framework [8] to identify attackers, analyze vulnerabilities through agent dependency links and suggest counter-measures. The main difference is that they start from agents involved in the system rather than goals being threatened. In contrast with our approach, they identify insider attackers only, that is, system stakeholders that were identified before in the primal model and might be suspect. The malicious goals owned by such attackers are not modelled explicitly. Their methodology provides no formal techniques for building threat models. The strength of their approach is the propagation of vulnerabilities along agent dependency links.

Dwyer and colleagues have built an extensive catalogue of formal specification patterns [14]. Their patterns are also based on temporal behaviors but not specific to security.

Sheyner et al use model checking technology to generate and analyze attack graphs [39]. Given a state machine model N of the network under attack, a model A of the attacker's capabilities and a desired security property S, their tools produce all possible counterexample scenarios found when trying to verify

$$N, A \models S$$

Our work may be seen as an "upstream" complement to their approach; it can be applied sooner in the process, to point out earlier security problems at the requirements level, by deriving threats through deductive inference from partial, declarative goal/anti-goal models - as opposed to operational state machine models that need to be completely built up before their technique can be applied.

8. Conclusion

Our focus in this paper was on security enginering at the application layer exclusively; compared with the crypto, protocol and system/language layers the application layer has received much less attention to date. As a prerequisite for security assurance at this layer, analysts must ensure that application-specific security requirements are made explicit, precise, adequate, non-conflicting with other requirements and complete.

We presented a requirements engineering method for elaborating security requirements based on the incremental building and specification of two concurrent models: an intentional model of the system-to-be and an intentional anti-model yielding vulnerabilities and capabilities required for achieving the anti-goals of threatening security goals from the original model. The method allows threat trees to be derived systematically through anti-goal refinement until leaf nodes are reached that are either attackee vulnerabilities observable by the attacker or anti-requirements implementable by this attacker. The original model is then enriched with new security requirements derived as countermeasures to the anti-model.

Our approach extends the KAOS framework for goal-oriented requirements engineering [11, 22, 23] in several ways:

- it extends the specification language with epistemic constructs for reasoning about the attacker's knowledge;
- it provides specification patterns for formal elicitation of candidate security requirements to start the analysis;
- it introduces a *duality principle* for richer modeling of threats; system goals, requirements, expectations about the environment and software services are transposed to malicious obstacles to security requirements, implementable anti-requirements, software vulnerabilities and attacker capabilities, respectively;
- it supports the obstacle likelihood and severity assessment phase by linking malicious obstacles to attackers and their malicious goals.

Our approach is intended to provide constructive guidance in early elaboration of security concerns; it supports incremental reasoning on partial models and formal derivation when higher assurance is needed; alternative threats and countermeasures may be modelled explicitly.

Although the original goal-oriented model building method has been validated extensively (over 25 industrial projects in a wide variety of domains), its transposition to anti-model building has been limited to four case studies so far: the one used for illustration in this paper, a popular smartcard-based payment system used in Belgium, a web-based CD order tracking system and an e-commerce system relying on an independent payment processing agent. In particular, a similar derivation from a confidentiality goal about order information allowed us to derive vulnerabilities found in reported attacks on a well-known CD sales company, namely, (a) confidential order information was obtainable just by submitting guessable order numbers, and (b) order numbers were appearing on the page URL displayed by the order tracking service. For the e-commerce system with independent payment processing, the anti-model revealed a man-in-the-middle attack making malicious use of the system's services; this was obtaind by anti-goal regression through the system's goals (in addition to domain properties, see Section 4). In the latter case study, the initial anti-goal was not a negated instantiation of a security goal pattern but just a negated functional goal ("the item is sent but not paid"). This case study also suggested the need for distinguishing vulnerabilities that are more critical in anti-goal achievement than others.

This leads us to a number of issues still left open, e.g.,

- When does the cycle "requirement/anti-requirement/ countermeasure" terminate?
- Can we build richer catalogues of threat patterns and corresponding "best" countermeasure patterns?
- To what extent do boundary conditions for conflict among multiple goals [20] play the role of covert channels that can be exploited by attackers to satisfy their anti-goals?
- How do we incorporate trust models in this framework and model trustworthy agents?
- Can we incorporate probabilistic frameworks to reason about *partial* satisfaction of goals/anti-goals and determine the likelihood of anti-goal occurrences?

These are issues we are working on.

Acknowledgement. This work was partially supported by the Regional Government of Wallonia (MILOS project, RW Conv. 114856). Clarifying discussions with Bashar Nuseibeh, Renaud De Landtsheer, David Janssens and Emmanuel Letier are gratefully acknowledged. Special thanks are due to Renaud and David for feedback from their experimentation of the method on the Tower Records and Paypal case studies, respectively.

References

[1] I. Alexander, "Misuse Cases: Use Cases with Hostile Intent", *IEEE Software*, Jan/Feb 2003, 58-66.

[2] E.J. Amoroso, *Fundamentals of Computer Security*. Prentice Hall, 1994.

[3] A. Anton, J. Earp and A. Reese, "Analyzing Website Privacy Requirements Using a Privacy Goal Taxonomy", *Proc. RE'02 – IEEE International Requirements Engineering Conference*, Essen, September 2002, 23-31.

[4] S. Brohez and Y. Grégoire, *Obstacle Monitoring: an Implementation based on the ASAX Intrusion Detection System*. M.S. Thesis, University of Namur, July 2002.

[5] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication", *ACM Transactions on Computer Systems*, Vol. 8 No. 1, Feb. 1990, 18-36.

[6] Common Criteria for Information Technology Security Evaluation, Aug. 1999, http:www.commoncriteria.org/.

[7] CERT, http://www.cert.org/stats/cert_stats.html.

[8] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic, Boston, 2000.

[9] E.M. Clarke, S. Jha and W. Marrero, “Verifying Security Protocoles with Brutus”, *ACM Trans. Software Engineering & Methodology* Vol. 9 No. 4, Oct. 2000, 443-487.

[10] R. Crook, D. Ince, L. Lin and B. Nuseibeh, “Security Requirements Engineering: When Anti-Requirements Hit the Fan”, *Proc. RE’02 – IEEE International Requirements Engineering Conference*, Essen, September 2002, 203-205.

[11] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-Directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[12] R. Darimont and A. van Lamsweerde, “Formal Refinement Patterns for Goal-Driven Requirements Elaboration”, *Proc. FSE’4 - Fourth ACM SIGSOFT Symp. on Foundations of Software Engineering*, San Francisco, Oct. 1996, 179-190.

[13] Proceedings of the DIMACS Workshop on *Design and Formal Verification of Security Protocols*. Rutgers University, September 1997.

[14] M.B. Dwyer, G. S. Avrunin and J.C. Corbett, “Patterns in Property Specifications for Finite-State Verification”, *Proc. ICSE’99 - 21st Intl. Conf. Software Engineering,* May 1999.

[15] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, “Reconciling System Requirements and Runtime Behaviour”, *Proc. IWSSD’98 - 9th Int.l Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[16] N. Habra, B. Le Charlier, A. Mounji, I. Mathieu, “ASAX: Software Architecture and Rule-base Language for Universal Audit Trail Analysis”, *Proc. 2nd European Symp. on Research in Computer Security (ESORICS’92),* Nov. 1992.

[17] G. Helmer, J. Wong, M. Slagell, V. Honavar , L. Miller and R. Lutz, “A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System”, *Requirements Engineering Jl*. Vol. 7 No. 4, 2002, 177-220.

[18] R. Kemmerer, C. Meadows, and J. Millen, "Three systems for Cryptographiuc Protocol Analysis", *Journal of Cryptology*, Vol. 7 No. 2, 1994, 79-130.

[19] R.A. Kemmerer, “Cybersecurity”, Invited Mini-Tutorial, *Proc. ICSE’03: 25th Intl. Conf. on Software Engineering,* Portland, IEEE CS Press, May 2003, 705-715.

[20] A. van Lamsweerde, R. Darimont, E. Letier, “Managing Conflicts in Goal-Driven Requirements Engineering”, *IEEE Transactions on Software Engineering,* Nov. 1998, 908-926.

[21] A. van Lamsweerde, “Requirements Engineering in the Year 00: A Research Perspective”, Keynote paper, *Proc. ICSE’2000 - 22nd Intl. Conf. on Software Engineering,* 2000.

[22] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering”, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, Vol. 26 No. 10, Oct. 2000, 978-1005.

[23] A. van Lamsweerde , “Goal-Oriented Requirements Engineering: A Guided Tour”, *Invited Minitutorial, Proc. RE’01 - 5th Intl. Symp. Requirements Engineering,* Toronto, August 2001, pp. 249-263.

[24] E. Letier and A. van Lamsweerde, “Agent-Based Tactics for Goal-Oriented Requirements Elaboration”, *Proc. ICSE’02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Computer Society Press, May 2002.

[25] E. Letier and A. van Lamsweerde, “Deriving Operational Software Specifications from System Goals”, *Proc. FSE’10:* 10th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[26] N. Leveson, *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[27] L. Lin, B. Nuseibeh, D. Ince, M. Jackson and J. Moffett, “Introducing Abuse Frames for Analyzing Security Requirements”, Internal Report, Open University, 2003.

[28] L. Liu, E. Yu and J. Mylopoulos, “Security and Privacy Requirements Analysis within a Social Setting”, *Proc. RE’03 – Intl. Conf. Requirements Engineering,* Sept. 2003, 151-161.

[29] G. Lowe, “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR”, *TACAS’96*, Springer LNCS 1055, 1996, 147-166.

[30] AP. Moore, R.J. Ellison and R.C. Linger, “Attack Modeling for Information Security and Survivability”, Technical Note CMU/SEI-2001-TN-001, March 2001.

[31] http://www.objectiver.com.

[32] D.L. Parnas and J. Madey, “Functional Documents for Computer Systems”, *Science of Computer Programming,* Vol. 25, 1995, pp. 41-61.

[33] C. Potts, “Using Schematic Scenarios to Understand User Needs”, *Proc. DIS’95 - ACM Symp. on Designing Interactive Systems,* Univ. Michigan, August 1995.

[34] A. Rifaut, P. Massonet, J.F. Molderez, C. Ponsard, P. Stadnik, H. Tran Van and A. van Lamsweerde, “FAUST: Formal Analysis of Goal-Oriented Requirements Using Specification Tools”, *Proc. RE’03,* Monterey, Sept. 2003, 350.

[35] W. N. Robinson, “Requirements Interaction Management”, *ACM Computing Surveys*, June 2003.

[36] A. dos Santos, G. Vigna, and R. Kemmerer, “Security Testing of the Online Banking Service of a Large International Bank”, *Proc. 1st Workshop on Security and Privacy in E-Commerce,* November 2000.

[37] B. Schneier, *Applied Cryptography*. Wiley, 1996.

[38] B. Schneier, *Secrets and Lies: Digital Security in a Networked World*. Wiley, 2000.

[39] O. Sheyner, J. Haines, S. Jha, R. Lippmann and J. Wing, “Automated Generation and Analysis of Attack Graphs”, *Proc. IEEE Symp. Security and Privacy*, Oakland, May 2002.

[40] G. Sindre and A.L. Opdahl, “Eliciting Security Requirements by Misuse Cases, *Proc. TOOLS Pacific’2000 - Techn. of Object-Oriented Languages and Systems*, 120-131.

[41] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way.*Addison-Wesley, 2001.

[42] J. Wing, "A Symbiotic Relationship Between Formal Methods and Security", *Proc. NSF Workshop on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solution.* December 1998.