Attack Patterns for Security Requirements Engineering

Laurent A. Hermoye and Axel van Lamsweerde
Département d'ingénierie informative
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
lahermoye@gmail.com, avl@info.ucl.ac.be

Dewayne E. Perry
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
perry@ece.utexas.edu

Abstract

The importance of security concerns at requirements engineering time is increasingly recognized. However, little support is available to help requirements engineers elaborate adequate, consistent, and complete security requirements. The paper presents a reuse-based approach for modeling, specifying, and analyzing application-specific security requirements. The method is based on a goal-oriented framework that addresses malicious goals (called anti-goals) set up by attackers to threaten security goals. Threat trees are built systematically through specializations of attack patterns. Such patterns abstract the attacker's anti-goals for known attacks. Once specialized to the specific vocabulary of the application, the patterns are further refined until fine-grained anti-goals are obtained that are either software vulnerabilities observable by the attacker or anti-requirements implementable by this attacker. New security requirements are then obtained as specializations of generic countermeasures associated with the attack pattern. The method is illustrated with an attack pattern for replay attacks and its specializations to an e-commerce system and a mailing system.

1. Introduction

Reports on software vulnerabilities have increased dramatically [2]. The technologies used by attackers make software systems increasingly insecure. Early identification of software vulnerabilities can thus have a major impact on security.

The security guarantee provided by current security technology is inversely proportional to the "size" of the software layer at which the technology applies [24]. At the bottom, the crypto layer offers solid and well-established techniques for basic services such as encryption/decryption. 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. Above the security layer, the system layer provides standard services, implemented in some programming language, such as remote file access; services like SSH or SSL, and language technologies like Active X or Java. The state of the art in these three layers tells us what can be guaranteed and what is still vulnerable. The top application layer offers services such as web-based banking operations that must implement application-specific security requirements in terms of facilities provided by the lower layers. The state of the art in security engineering at the application layer is much more limited [24]. This paper focusses on security engineering 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 made 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. Two models can be built in parallel to anticipate such attack scenarios: (1) a goal refinement graph for the system-to-be that covers both the software and its environment, (2) a threat graph, derived from the goal graph, that exhibits how the system goals could be maliciously threatened, why and by whom [21].

Building such threat models is a fastidious process. As it is the case for other software artefacts, such process might be made easier and more cost-effective if we can partly reuse existing threat models to produce new ones [15]. The current build-from-scratch techniques might benefit from techniques that emphasize threat model construction from reusable patterns.

The paper focusses on the reuse of threat models and countermeasures in the context of engineering application-specific security requirements. Our approach consists in defining reusable threat trees, abstracted from known attacks such as replay, denial-of-service, or password attacks.

Once specialized to the specifics of the application, the pattern is used to systematically derive malicious goals. The resulting threat tree is then further refined until anti-requirements are derived that match the attacker's capabilities.
Reusable countermeasures are associated to the patterns. Their corresponding specialization to the application produces new requirements as responses to the exposed anti-requirements and vulnerabilities.
The paper is organized as follows. Section 2 provides some necessary background. Section 3 introduces attack patterns as reusable models for threat analysis. Section 4 details our approach on one pattern for replay attacks and its specialization to two applications (an e-commerce system and a mine pump system). Section 5 discusses the current coverage of our pattern catalog. Section 6 reviews related work whereas Section 7 summarizes and evaluates our contribution.

2. Background

A *goal* is a prescriptive statement of intent whose satisfaction requires the cooperation of some of the agents forming the system [20]. Goals capture the rationale of the envisioned system; they make it explicit *why* a new or modified system is needed. Goals are operationalized into specifications of services and constraints. These specifications refer to *what* the system should do and *how well*. The responsibilities for fine-grained goals are to be assigned to individual agents such as humans, devices and software. This *who* dimension results in the definition of the boundary between the software and the environment. Goals are organized in hierarchies where high-level goals are coarse-grained (e.g., "secure transaction") and low-level goals are fined-grained (e.g., "card swallowed after 3 wrong attempts"). Goals are elicited through a refinement/abstraction process. An AND-refinement links a goal to a set of subgoals. Satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. An OR-refinement links a goal to alternative subgoals. Satisfying one of the subgoals is a sufficient condition for satisfying the goal. The specification of a goal can be formally expressed in a real-time linear temporal logic using the following temporal operators,

$\Diamond$ *(eventually)* $\quad$ $\blacklozenge$ *(some time in the past)*
$\Box$ *(always in the future)* $\quad$ $\blacksquare$ *(always in the past)*

and the following epistemic operators [21],

$KnowsV_{ag}(v) \equiv \exists x : Knows_{ag}(x = v)$ *(knows value)*
$Knows_{ag}(P) \equiv Belief_{ag}(P) \wedge P$ *(knows property)*

where $Belief_{ag}(P)$ means P is among the properties stored in the local memory of agent ag. For example, an obvious goal for a mailing system would be the following:

Goal *Achieve*[EmailReceivedInTime]
Refines EffectiveMailingSystem
RefinedTo EMailAtServerInTime, EMailDispatchedInTime, EMailDeliveredInTime
FormalDef $\forall s : Sender, e : eMail, r : Receiver$
$Sends(s, e, r) \Rightarrow \Diamond_{\leq d} Receives(r, e)$

An object model can be derived systematically from the goal model as a UML class diagram. In our example it would capture agents such as *Sender*, *Receiver*, entities such as *eMail*, associations such as *Sends*, *Receives*, and corresponding domain properties as object invariants. Goals can be refined through instantiations of refinement patterns [4]. The refinement process ends when every goal is realizable by an agent.
An *obstacle* to some goal is a condition whose satisfaction may prevent the goal from being achieved [22]. A condition O is said to be an obstacle to goal G if :

1. $\{O, Dom\} \vdash \neg GA$ (obstruction)
2. $Dom \nvdash \neg O$ (domain consistency)
3. O is realizable by agents not involved in G

Obstacle analysis consists in identifying obstacles by regressing goal negations through the set of available domain properties until they are assignable to agents [22]. Such regression consists in calculating preconditions for obtaining the negation of the goal assertion in view of what is known about the domain. For example, the obstacle UnreachableServer is easily derived by regression of the negation of the goal EmailReceivedInTime previously defined. Obstacles are resolved using resolution operators [22], to produce more robust requirements (e.g., introducing a backup server if the main server is out of service).
Obstacle analysis appears too limited for analyzing attacks to satisfy malicious goals based on the attacker's capabilities and the system's vulnerabilities [21]. Threat models are goal refinement graphs where the goals are malicious goals (called anti-goals). Threat analysis based on anti-goals consists in (1) negating specification patterns for security goals instantiated to sensitive attributes/associations from the object model, (2) elaborating an anti-goal AND/OR refinement graph until attacker capabilities and system vulnerabilities are reached, and (3) deriving alternative countermeasures to the threats found. Let us illustrate this in the e-mailing example. The generic goal

GoalPattern *Avoid*[SensitiveInfoKnownByUnauthorized]
FormalDef $\forall a : Agent, ob : Object$
$\neg Authorized(a, ob.Info) \Rightarrow \neg KnowsV_a(ob.Info)$

is instantiated and negated which yields

AntiGoal *Achieve*[EmailKnownByThirdParty]
FormalDef $\exists a : Attacker, e : eMail$
$\neg isReceiverOf(a, e.Info) \wedge KnowsV_a(e.Info)$

A Thief, Spy, or Terrorist agent could benefit from the above

anti-goal. A regression and resolution of the anti-goal are illustrated in Figure 1.

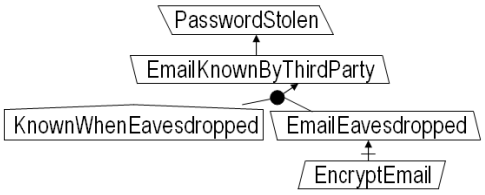

Figure 1. Anti-model for the e-mailing system

3. Attack Patterns

The process of building threat models should be supported by techniques and tools. One technique is to reuse threat models built for known attacks. Commonalities among such attacks are analyzed from reports and case studies. The corresponding anti-goals are abstracted and linked through refinement/contribution links to form an attack pattern. The resulting threat model fragment captures common objectives of malicious agents for known attacks, that can be reused in multiple contexts. Once specialized to an application-specific vocabulary, the attack pattern may provide useful domain properties and anti-goals. The anti-goals in the specialized fragment should then be further refined until leaf nodes are derived that are either software vulnerabilities observable by the attacker or anti-requirements that are realizable by the attacker in view of his/its capabilities. The higher-level anti-goals capture the rationale for the attack.

3.1 Attack Patterns as Anti-goal Graphs over Abstract Domains

Several concepts are important for understanding and defining attack patterns.
- A *specialization* is a binary relation between two concepts. IsA(A, B) means that A is a specialization of B. IsA(A,B) holds iff every instance of concept A is also an instance of concept B and there are instances of concept B that are not instances of concept A.
- An application *domain* is a concept aggregating finer-grained concepts that share a common, application-specific vocabulary.
- An application domain can be more *abstract* or more *concrete* than others. A domain is more concrete than another if the former is a specialization (isA) of the latter that brings specific knowledge from its application area [10]. In general, a more abstract domain subsumes multiple more concrete domains that specialize it. For example, the Plane and Train domains are specializations of the Transportation domain.
- *Abstract anti-goals, domain properties*, and *predicates* are reusable concepts defined on abstract domains. They are to be specialized to concrete domains at reuse time.
- An *attack pattern* is a fragment of an anti-goal graph defined on some abstract domain. It is built in terms of abstract anti-goals and abstract domain properties. The assertions defining them are formalized in linear temporal logic using abstract predicates. At reuse time, the abstract predicates need to be instantiated to the relevant concrete domain in order to produce a specialized, application-specific fragment of anti-goal graph. Such instantiation is performed by replacing the meta-variables occurring in the abstract predicate by application-specific ones that correspond to them in the considered specialization.

For example, the attack pattern for denial-of-service attacks (DoS) states that the attacker wants some resource needed by an agent to be unusable (see top of Figure 2). The anti-goal refinement provides a means for making the resource unusable by making it unavailable. The attack pattern is specialized in two concrete domains: an e-commerce domain and an air traffic domain. The first specialization

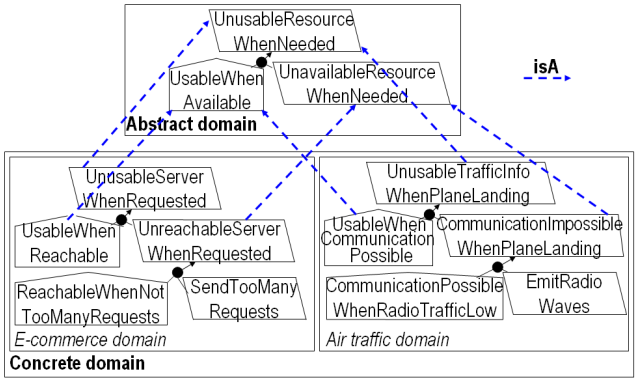

Figure 2. Attack Pattern for *DoS* Attacks

states that the attacker wants a server to be unusable by known clients. The specialized refinement provides a means for making the server unusable by sending many requests. The second specialization states that the attacker wants traffic information sent by the traffic tower to be unusable when a plane is landing. The specialized refinement provides a means for making the communication impossible by emitting disrupting radio waves.

Attack patterns are built systematically by regression of abstract anti-goals through abstract domain properties. For example, consider the following abstract anti-goal:

AntiGoal *Achieve*[UnusableResourceWhenNeeded]
FormalDef $\Diamond\exists u : User, r : Resource$
$Needs(u, r) \wedge Authorized(u, r) \wedge$
$\Box_{<M} \neg Using(u, r)$

In the abstract domain of resource management, there is a

domain property saying that resource availability is a necessary condition for resource usage:

DomProp [UsableWhenAvailable]
FormalDef $\forall u : User, r : Resource$
$Using(u, r) \Rightarrow Available(r)$

A single one-step regression [22] of the above anti-goal UnusableResourceWhenNeeded through the abstract property UsableWhenAvailable yields the following abstract subgoal:

AntiGoal *Achieve*[UnavailableResourceWhenNeeded]
FormalDef $\Diamond\exists u : User, r : Resource$
$Needs(u, r) \wedge Authorized(u, r) \wedge$
$\Box_{<M} \neg Available(r)$

We can now specialize the attack pattern to an e-commerce system and to an air traffic system as follows.

- **E-Commerce:** *Client* isA *User*, *Server* isA *Resource*, *Requests* isA *Needs*, *KnownClient* isA *Authorized*, *UsingServer* isA *Using*, and *Reachable* isA *Available*.
- **Air traffic:** *Plane* isA *User*, *TrafficInfo* isA *Resource*, *NeedsForLanding* isA *Needs*, *isEmitterOfKnownWaves* isA *Authorized*, *Communicating* isA *Using*, and *CommunicationPossible* isA *Available*.

For the e-commerce system, for example, the concrete anti-goal UnusableServerWhenRequested is derived as a specialization of the abstract anti-goal UnusableResourceWhenNeeded:

AntiGoal *Achieve*[UnusableServerWhenRequested]
Specializes UnusableResourceWhenNeeded
FormalDef $\Diamond\exists c : Client, s : Server$
$Requests(c, s) \wedge KnownClient(c, s) \wedge$
$\Box_{<M} \neg UsingServer(c, s)$

The abstract anti-goal is similarly specialized to the air traffic system. Other abstract anti-goals and abstract domain properties are specialized similarly.

3.2 Documenting Attack Patterns

Attack patterns are made from abstract anti-goals and domain properties. They are documented in a pattern catalog as follows.

- **Attack context**
  Here come the circumstances in which the attacker might be able to compromise a system. They are captured as abstract security goals that the attacker might consider as a target. For example, UsableResourceWhenNeeded is such an abstract goal:
  Goal *Achieve*[UsableResourceWhenNeeded]
  FormalDef $\forall u : User, r : Resource$
  $Needs(u, r) \wedge Authorized(u, r) \Rightarrow Using(u, r)$
  It is indeed the case that an attacker may want a resource to be unusable by an authorized user needing it.
- **Vocabulary**
  The concepts used in the attack pattern are defined here. These include entities, agents, relationships, events, operations, etc.
- **Pattern building process**
  Here come the informal and formal definitions of the abstract anti-goals and domain properties found in the attack pattern. This section also describes the regression used for deriving lower-level anti-goals from higher-level ones through the abstract domain properties.
- **Generic Countermeasures to threat**
  This section lists alternative anti-goal resolutions based on known tactics.

3.3 Using Attack Patterns

In the process of elaborating the requirements for her application, the user has to retrieve relevant attack patterns, specialize them to the application, adapt the result wherever appropriate, expand the result to get an entire threat model, and produce new requirements from the instantiated countermeasures.

- **Retrieve relevant patterns**
  The patterns whose root anti-goal negates a relevant abstract security goal are selected. The root anti-goals are to be specialized to the sensitive objects of the application's object model. For example, UnusableResourceWhenNeeded negates UsableResourceWhenNeeded which proves to be relevant. It can be specialized to UnusableServerWhenAccessed (*Server* isA *Resource*).
- **Specialize matching concepts and adapt the result**
  Each meta-variable in the abstract predicates found in a selected attack pattern is instantiated to the specialized concept matching it in the application. The resulting application-specific anti-goal tree is adapted if necessary. A specialized concept P matches an abstract concept X if the definition of X covers the definition of P, that is, the set of instances corresponding to the definition of X is a superset of the set of instances corresponding to the definition of P. This condition is called the $[X/P]$ matching condition.
  For example, let us consider the following definitions:
  - **Resource:** An entity that an agent may request for some usage [3].
  - **Server:** A device that a client may request for obtaining network services.
  - **TrafficInfo:** Radio waves that an aircraft is using to communicate with air traffic controllers.

The concept of *Resource* is defined on an abstract domain. Specializations include a resource used for network services or for aircraft-controller communication. In the e-commerce domain, the matching condition $[Resource/Server]$ holds because the definition of *Resource* covers the definition of *Server*. In the air traffic domain, the matching condition $[Resource/TrafficInfo]$ holds because the definition of *Resource* covers the definition of traffic information being used for communication between an aircraft and air traffic controllers.

- **Expand the specialized threat tree**
  The specialized anti-goal tree fragment is further abstracted and refined.
  - *WHY* questions are asked to find out the higher-level motivations of the attacker.
  - *HOW* questions are asked to progressively reach anti-requirements that are realizable by the attacker in view of his capabilities and software vulnerabilities that are observable by the attacker in view of those capabilities.
- **Derive new requirements from the specialized countermeasures**
  The generic countermeasures proposed in the attack pattern are specialized according to the matching concepts, their adequacy as new application-specific security requirements is assessed, and the adequate security requirements are integrated in the application's goal graph.

4. Example: Attack Pattern for Replay Attacks

Attack patterns can be defined for every type of attack. To demonstrate our approach, we detail replay attacks as an example. These are attacks on an authentication system by recording and replaying previously played information [8]. We discuss this pattern following the structure proposed in Section 3.2 and Section 3.3.

4.1 Documenting the Replay Attack Pattern

4.1.1. Attack Context

The context in which an attacker might have a malicious intention is first introduced with a definition:

Def A message is said to be *fresh* if it was never sent in the past.
FormalDef $\forall m : Mess,\ \ Fresh(m) \triangleq \neg\exists s : Sender, r : Receiver,\ \ \blacklozenge Sends(s, m, r)$

The abstract goal that attackers may want to compromise is the following:

Goal *Maintain*[UnfreshMessageNeverAccepted]
Specializes ReplayedObjectNeverAccepted
Def An unfresh message should never be accepted by a receiver.
FormalDef $\forall m : Mess,\ \ \neg Fresh(m) \Rightarrow \Box\neg\exists r : Receiver,\ Accepts(r, m)$

4.1.2. Vocabulary

The concepts used to define the replay attack pattern include the following:
Channel: Channel through which agents communicate.
Fresh: Never sent in the past.
Accepts: Message acceptance by some receiver.
Records: Recording, by some attacker, of a message from a sender to a receiver.
Listens: Message listening on some channel.
Forges: Message transformation to make it acceptable even if it is not fresh.

4.1.3. Pattern Building Process

The attack pattern for replay attacks (see Figure 3) is inspired from the taxonomy proposed by Syverson [19]. It covers the list of replay attacks proposed by Gong [6].

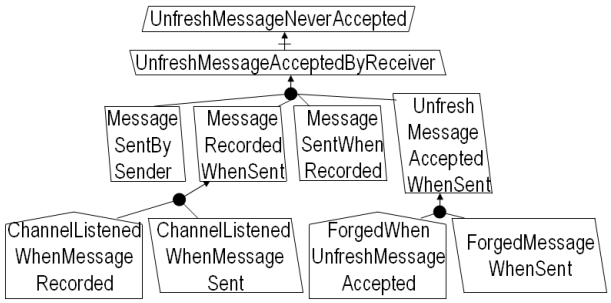

Figure 3. Attack Pattern for Replay Attacks

The root of the attack pattern is the negation of the abstract goal UnfreshMessageNeverAccepted :

AbsAntiGoal *Achieve*[UnfreshMessAcceptedByReceiver]
Def An unfresh message is accepted by some receiver.
FormalDef $\Diamond\exists m : Mess, r : Receiver\ \neg Fresh(m) \wedge \Diamond Accepts(r, m)$

If an attacker wants to replay a message that is accepted by some receiver, (G1) a legitimate sender has to send a message to some receiver, (G2) the attacker has to record that message, (G3) the attacker has to replay the message to some receiver, (G4) the receiver has to accept the replayed message. These four goals define milestones for an attacker to successfully replay a message. Instantiating the milestone pattern in [9] to the sensitive objects in the replay context, we obtain the following formal refinement of the

abstract anti-goal UnfreshMessageAcceptedByReceiver :

AntiGoal *Achieve*[MessageSentBySender]
Def A message has to be sent by a sender to some receiver.
FormalDef $\Diamond\exists m : Mess, s : Sender, r : Receiver,\ Sends(s, m, r)$

AntiGoal *Achieve*[MessageRecordedWhenSent]
Def The message has to be recorded by an attacker when it is sent by a sender to some receiver.
FormalDef $\forall m : Mess, s : Sender, r : Receiver\ Sends(s, m, r) \Rightarrow \Diamond\exists a : Attacker\ Records(a, m, r)$

AntiGoal *Achieve*[MessageSentWhenRecorded]
Def The attacker has to send the message to the receiver after he recorded it.
FormalDef $\forall m : Mess, a : Attacker, r : Receiver\ Records(a, m, r) \Rightarrow \Diamond Sends(a, m, r)$

AntiGoal *Achieve*[UnfreshMessAcceptedWhenSent]
Def The message should be unfresh and accepted when the attacker sent it to the receiver.
FormalDef $\forall m : Mess, a : Attacker, r : Receiver\ Sends(a, m, r) \Rightarrow \neg Fresh(m) \wedge \Diamond Accepts(r, m)$

Let us now refine the abstract anti-goal MessageRecordedWhenSent by regression. A necessary and sufficient condition for the target predicate $Records(a, m)$ is that the attacker listens messages on a corresponding channel. This yields the following abstract domain property:

DomProp ChannelListenedWhenMessRecorded
FormalDef $\forall m : Mess, r : Receiver, a : Attacker\ Records(a, m, r) \Leftrightarrow \exists c : Channel\ Listens(a, m) \wedge On(m, c)$

A one-step regression [22] of MessageRecordedWhenSent through ChannelListenedWhenMessageRecorded yields the following abstract anti-goal:

AntiGoal *Achieve*[ChannelListenedWhenMessSent]
Def The communication channel must be listened when the sender sends a message to the receiver.
FormalDef $\forall m : Mess, s : Sender, r : Receiver\ Sends(s, m, r) \Rightarrow \Diamond\exists a : Attacker, c : Channel\ Listens(a, c) \wedge On(m, c)$

The abstract anti-goal UnfreshMessageAcceptedWhenSent is further refined by regression. A necessary and sufficient condition for acceptance of an unfresh message is that the attacker can transform the message so that it is accepted by a receiver :

DomProp ForgedWhenUnfreshMessageAccepted
FormalDef $\forall m : Mess, r : Receiver\ \neg Fresh(m) \wedge \Diamond Accepts(r, m) \Leftrightarrow \exists a : Attacker,\ \blacklozenge Forges(m, a)$

A one-step regression of UnfreshMessageAcceptedWhenSent through ForgedWhenUnfreshMessageAccepted yields the following abstract anti-goal:

AntiGoal *Achieve*[ForgedMessageWhenSent]
Def The message had to be forged by the attacker when it is sent to the receiver.
FormalDef $\forall m : Mess, a : Attacker, r : Receiver\ Sends(a, m, r) \Rightarrow \blacklozenge Forges(m, a)$

4.1.4. Generic Countermeasures to Threat

Countermeasures against replay attacks include the following (see Figure 4):

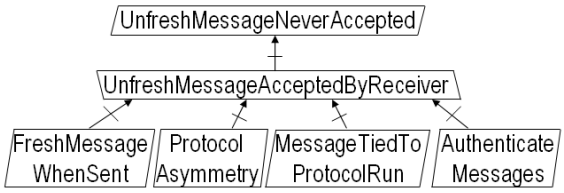

Figure 4. Generic Countermeasures for Replay Attacks

- Introduce a freshness mechanisms (e.g., messages should be tied with a unique identifier),
- Introduce asymmetry (e.g., avoid a man-in-the-middle attack due to protocol symmetry),
- Tie messages to a particular use at a particular time (e.g., messages should be tied to a particular protocol run rather than a particular epoch. Messages from different protocol runs would therefore be revealed),
- Authenticate the message's emitter and recipient (e.g., cryptographically bind the name of a message originator to the message).

4.2 Reusing the Replay Attack Pattern

The attack pattern for replay attack can be specialized to different application domains. We illustrate this by taking two completely different domains: an e-commerce domain (Section 4.2.1) and to a mine pump domain (Section 4.2.2). The Replay Attack pattern was reused in several other domains [9] omitted here for space reasons.

4.2.1. Pattern Reuse in an E-Commerce System

Consider the ordering procedure in a e-commerce system. A client orders a product via a transaction sent to the account manager. The account manager then chooses to commit or to reject the sent transaction. One objective is to

avoid that transactions are committed twice. A malicious attacker might want to compromise that goal by replaying a modified transaction so that a known client pays for his transaction.

4.2.1.1. Retrieve Relevant Pattern

The contextual security goal is obtained by specializing the abstract goal UnfreshMessageNeverAccepted to the e-commerce domain:

Goal *Maintain*[OldTransactionNeverCommitted]
Specializes UnfreshMessageNeverAccepted
Def An old transaction should never be committed by the transaction manager.
FormalDef $\forall t : Trans,\ \neg New(t) \Rightarrow \Box\neg\exists tm : TrManager,\ Commits(tm, t)$

The definition of freshness is specialized to the e-commerce domain accordingly:

Def A transaction is said to be *new* if it was never used for a client order in the past.
FormalDef $\forall t : Trans,\ New(t) \triangleq \neg\exists c : Client, tm : TrManager,\ \blacklozenge Orders(c, t, tm)$

4.2.1.2. Specialize Matching Concepts and Adapt the Result

The following specializations satisfy the matching condition:

[Channel/Network]
[Accepts/Commits]
[Records/Sniffes]
[Listens/Eavesdrops]
[Fresh/New]
[Forges/ModifiesTransactionNumber]

The threat model fragment specialized from the attack pattern of Figure 3 is illustrated in Figure 5.

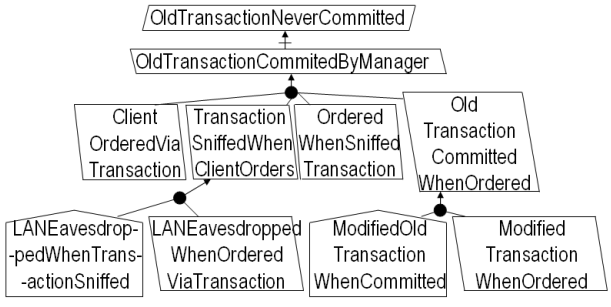

Figure 5. Specialized Attack Pattern

The correctness of attack patterns is established once for all. [9]. The regressions performed on the abstract model do not need to be replayed on the specialized one, of course; they are implicitly obtained by simple instantiation of the corresponding meta-variables. For example, consider the specialized anti-goal:

AntiGoal *Achieve*[TransSniffedWhenClientOrder]
Specializes MessageRecordedWhenSent
Def Transactions that the client sent for ordering should be sniffed.
FormalDef $\exists t : Trans, c : Client, tm : TrManager\ Orders(c, t, tm) \Rightarrow \Diamond\exists a : Attacker,\ Sniffs(a, t)$

The regression of this specialized anti-goal through the following specialized domain property is obtained for free:

DomProp LANEavesdroppedWhenTransactionSniffed
Specializes ChannelListenedWhenMessageRecorded
Def An attacker eavesdrops a LAN where transactions are transmitted iff he sniffs the transactions.
FormalDef $\forall t : Trans, a : Attacker\ Sniffs(a, t) \Leftrightarrow \exists l : LAN\ Eavesdrops(a, l) \wedge TransmittedOn(t, l)$

The reused regression of TransactionSniffedWhenClient-Order through LANEavesdroppedWhenTransactionSniffed yields the following formal specification:

AntiGoal *Achieve*[LANEavesdropWhenOrderedViaTrans]
Specializes ChannelListenedWhenMessageSent
FormalDef $\forall t : Trans, c : Client, tm : TrManager\ Orders(c, t, tm) \Rightarrow \Diamond\exists l : LAN, a : Attacker\ Eavesdrops(a, l) \wedge TransmittedOn(t, l)$

4.2.1.3. Expand the specialized threat tree

The produced threat model fragment must now be completed. We need to understand *why* an attacker may want to replay a transaction, *how* he would do it. Answering those questions will result in parent goals and subgoals, respectively, to complete the anti-goal graph.

- ***Why?*** An attacker may want the client to pay multiple times (e.g., for revenge or to smeer the reputation of e-commerce). He may also want another client to pay his orders.
- ***How?*** A LAN machine normally accepts messages only destined for itself. However, when the machine is in promiscuous mode, it may read all information regardless of destination. This enables the attacker to read transactions on the LAN.

4.2.1.4. Derive new requirements from the specialized countermeasures

Specialized countermeasures to consider for reuse might include the following:

- Enforce transactions to be certified by a third party so that the client is sure that the transaction is not replayed by an attacker;
- Forbid promiscuous modes on the LAN;

- Bind transactions to time and protocol run.

4.2.2. Pattern Reuse in a Mine Pump System

Consider a system to control water level in a mine. The pump is activated when the water level is too high and disactivated when the water level is too low. An alarm should also be raised when the gas level is too high. A sensor monitoring the environment has to send signals to the software controller so that it can raise the alarm when needed. One of the objectives is to make sure that sensor signals are raised only when critical based on the current gas level.

4.2.2.1. Retrieve Relevant Pattern

The above objective specializes the abstract goal UnfreshMessageNeverAccepted as follows.

Goal *Maintain*[AlarmNeverRaisedByOldSignal]
Specializes UnfreshMessageNeverAccepted
Def A signal that does not reflect the current gas level should never raise the alarm.
FormalDef $\forall s : Signal,\ \neg Current(s.GasLevel) \Rightarrow \Box\neg\exists c : Controller,\ RaisesAlarm(s, c)$

The definition of freshness is specialized to the mine pump domain accordingly:

Def A signal is said to be *current* if it was not sent in the past.
FormalDef $\forall si : Signal,\ Current(s.GasLevel) \triangleq \neg\exists s : Sensor, c : Controller\ \blacklozenge Signals(s, si, c)$

4.2.2.2. Specialize Matching Concepts and Adapt the Result

The following specializations satisfy the matching condition:

[Channel/Wire]
[Accepts/RaisesAlarm]
[Records/Saves]
[Listens/CapturesSignalsOn]
[Fresh/Current]
[Forges/ModifiesTime]

The threat model fragment specialized from the attack pattern in Figure 3 is illustrated in Figure 6. Refinements and regressions are specialized accordingly.

4.2.2.3. Expand the specialized threat tree

The produced threat model fragment must be completed. We must answer *why* and *how* questions to obtain a full threat model.

- ***Why?*** An attacker may want to replay a high gas level for multiple reasons: the mine to make losses, employees to have a break, safety to be compromised, etc.

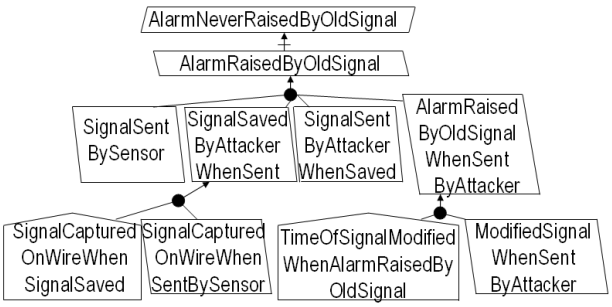

Figure 6. Specialized Attack Pattern

- ***How?*** The attacker may compromise the controller or intercept signals on the wire between the sensor and the controller.

4.2.2.4. Derive new requirements from the specialized countermeasures

Specialized countermeasures to consider for reuse might include the following:

- Instal the controller in a secure location;
- Make wiring inaccessible;
- Introduce freshness mechanism;
- The sensor might authenticate itself when it sends a signal.

5. A Catalog of Attack Patterns

Attack patterns can be built by abstracting various types of attacks known from the literature, e.g., [23], and by specifying them according to the structure described in Section 3.2. Grouping these patterns in a catalog provides a systematic basis for identifying malicious goals, plans to achieve such goals, system vulnerabilities, and countermeasures to introduce as new requirements. Multiple patterns can be composed within the same application.
The other patterns we have built, specified, and used on multiple case studies so far include the following [9]:

- An attack pattern for denial-of-service attacks captures 13 reusable anti-goals and their associated countermeasures. For example, the anti-goal TooManyAllocationsWhenResourceRequestedByAuthorized can be specialized to BandwidthOverflowWhenServerRequestedByClient (e-commerce system), and to TooManyEmailsInMailBoxWhenNewEmailArrived (e-mailing system).
- An attack pattern for password attacks covers a wide spectrum from the classical dictionary attack to social engineering attacks. The generic countermeasures

range from creating strong passwords to training naive employees.

- An attack pattern for critical stimulus/response systems captures anti-goals such as, e.g., InaccurateStimulusWhenCritical. This anti-goal can be specialized to the anti-goal SensorLevelLowWhenSystemLevelHigh in some systems, resulting in the alarm not being raised when the critical level is reached. In this specialization, the stimulus is the reaching of a high level of some dangerous item, and the response is to raise the alarm. The same abstract anti-goal can also be specialized to PulseSensorUnchangedWhenPlaneOnRunaway in an autopilot system, resulting in reverse thrust not being enabled. In this specialization, the stimulus is a signal from the pulse sensor and the response is to enable the reverse thrust.

- An attack pattern for masquerading attacks in which the attacker masquerades some resource to obtain confidential information. Specializations include the installing of a fake login screen to obtain passwords or fake atm screens to obtain secret codes or keyloggers.

6. Related Work

Attack trees [18] provide a convenient way of describing attacks on a system. The root node corresponds to some threat; child nodes capture ways of causing the parent node. Annotations can be assigned to nodes (e.g., likely/unlikely, easy/difficult, expensive/inexpensive, intrusive/nonintrusive). There are several differences with our technique. (a) Attack trees are not goal-anchored; they make it difficult to reason about malicious intentions. (b) No support is provided for identifying the root nodes in a threat tree. (c) No formal apparatus is available for deriving the causing child nodes in a systematic way. (d) The threats represented in a tree are application-specific and thus not meant to be reused.

The HAZOP technique relies on a set of guidewords for identifying failure nodes in risk trees [16]. Such guidewords should be applied to all important parameters of the process under study. For example, the guideword MORE means that there is more of some relevant quantity than there should be. Each guideword has specializations for particular application domains. Such specializations are called deviations. In contrast with our technique, (a) hazards are not regressed through reusable properties; (b) their rationale is not captured.

Desirable properties for process control systems can be listed in safety checklists to be reused from one application to the other [12]. If the properties are met by the current requirements, the system is said to be safe. If not, criteria should be determined to imply the unsatisfied properties. Properties and their corresponding criteria are reusable. They are intended to be specialized to application-specific versions. The main difference between safety checklists and attack patterns is that (a) checklists are not structured nor expressed in a form amenable to formal reasoning, and (b) malicious intentions cannot be captured.

Syverson defined a taxonomy of countermeasures against replay attacks [19]. The taxonomy is said to be complete because every type of replay attack can be resolved using one of the reusable countermeasures. The generic countermeasures in Section 4 are similar to Syverson's.

Threat descriptions were composed with a representation of functional requirements based on problem frames [11] in order to define system vulnerabilities [7]. The latter are ameliorated by security requirements expressed as constraints on the functional requirements. The main difference between this technique and attack patterns is that (a) threat descriptions are not meant to be reused, (b) they are informally defined, (c) there is no model of attacker agents, (d) there is no representation of the underlying malicious intentions, and (e) there is no support for threat identification and refinement.

Massonet describes an analogical reuse technique whereby goal-based requirements models are reused along a specialization hierarchy of abstract domains [17]. Several efforts were also made to identify domain-specific analysis patterns that can be reused from one application to the other within that domain. For example, Konrad and Cheng describes reusable UML diagrams for process control systems [14] whereas Fowler describes reusable class diagrams for a variety of management information systems [5].

7. Conclusion

Security is a big concern from the earliest stages of the software lifecycle. The definition of an adequate, coherent, and complete set of precise security requirements, specific to the application, is a difficult task for which little support is available. The paper has presented a formal reuse-based approach aimed at providing some guidance in this task. The requirements engineer may browse through a catalog of attack patterns, with associated countermeasures, determine which patterns might be relevant to the application, specialize the selected ones, and adapt and expand them as necessary. The patterns are expressed as refinement trees that exhibit intentional ways of breaking security goals. Commonalities among malicious goals, vulnerabilities, and domain properties were abstracted from the literature and from multiple case studies. Those commonalities were captured in intentional models defined on abstract domains. These models can be reused systematically for threat analysis by specializing them to the matching concepts in the application. Such specialization has to meet a precise criterion for

concept matching.

Each pattern is documented in a structured way that highlights its rationale for use, the process according to which it was built, and generic countermeasures. Such information is important for understanding the pattern, for assessing its applicability in specific situations, and for adapting its specializations to make them more adequate.

The attack patterns elaborated so far cover, on abstract domains, replay attacks,denial-of-service attacks, masquerading attacks, password attacks, and the corruption of accurate information required in critical stimulus-response systems. The catalog is obviously not intended to be comprehensive at this stage. Instead, we focussed on experimentation of those patterns through multiple specializations in very different domains. These include a realistic e-commerce system [1], a mine pump system [13], an e-mail system, and an air traffic system.

Our patterns were built using refinement and obstruction patterns proved correct once for all [22]. The underlying mathematics for establishing refinement completeness are kept hidden, and the proof is implicitly reused at specialization time.

While patterns may prove helpful in increasing the robustness and completeness of application-specific security requirements, there is no guarantee of course that they can be meaningfully applied in any domain for any context. Other forms of security analysis should complement them in the requirements engineer's palette.

Further work needs to focus on increasing the coverage of the catalog, assessing its effectiveness on a wider scale, and providing tool support for the retrieve, specialize, adapt, and expand stages of the reuse process described in the paper. A first step for increased coverage would be to define additional attack patterns for each security goal specification pattern in [21].

References

[1] S. Brohez. The RDS records inc. MILOS research case study. Technical report, Université Catholique de Louvain, June 2004.

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

[3] R. Darimont. *Process Support for Requirements Elaboration*. Phd Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, 1995.

[4] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190, 1996.

[5] M. Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[6] L. Gong. Variations on the themes of message freshness and replay – or the difficulty of devising formal methods to analyze cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 131–136, Los Alamitos, California, 1993. IEEE Computer Society Press.

[7] C. B. Haley, R. C. Laney, and B. Nuseibeh. Deriving security requirements from crosscutting threat descriptions. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 112–121, New York, NY, USA, 2004. ACM Press.

[8] N. Haller and R. Atkinson. On internet authentication, October 1994.

[9] L. A. Hermoye. Master Thesis, A Reuse-Based Approach To Security Requirements Engineering, June 2006.

[10] N. Iscoe, G. B. Williams, and G. Arango. Domain modeling for software engineering. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 340–343, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[11] M. Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[12] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. Softw. Eng.*, 17(3):241–258, 1991.

[13] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall, 1995.

[14] S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 127–136, Washington, DC, USA, 2002. IEEE Computer Society.

[15] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

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

[17] P. Massonet and A. van Lamsweerde. Analogical reuse of requirements frameworks. In *RE-97 - 3rd Int. Symp. on Requirements Engineering*, pages 26–37. IEEE, 1997.

[18] B. Schneier. *Secret and Lies: Digital Security in a Networked World*. John Wiley and Sons, 2000.

[19] P. Syverson. A taxonomy of replay attacks. In *Proceedings of the Computer Security Foundations Workshop VII, Franconia NH*, Los Alamitos, CA, USA, 1994. IEEE CS Press.

[20] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE01 - 5th Intl. Symp. Requirements Engineering*, pages 249–263. IEEE, 2001.

[21] A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 148–157, Washington, DC, USA, 2004. IEEE Computer Society.

[22] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, 2000.

[23] J. Whittakker and H. Thompson. *How to Break Software Security*. Pearson, 2004.

[24] J. Wing. A symbiotic relationship between formal methods and security, 1998.