Requirements Eng (2010) 15:41–62
DOI 10.1007/s00766-009-0090-z

SPECIAL ISSUE - SECURITY REQUIREMENTS ENGINEERING

A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities

Golnaz Elahi · Eric Yu · Nicola Zannone

Received: 29 October 2008 / Accepted: 26 October 2009 / Published online: 17 November 2009

Abstract Many security breaches occur because of exploitation of vulnerabilities within the system. Vulnerabilities are weaknesses in the requirements, design, and implementation, which attackers exploit to compromise the system. This paper proposes a methodological framework for security requirements elicitation and analysis centered on vulnerabilities. The framework offers modeling and analysis facilities to assist system designers in analyzing vulnerabilities and their effects on the system; identifying potential attackers and analyzing their behavior for compromising the system; and identifying and analyzing the countermeasures to protect the system. The framework proposes a qualitative goal model evaluation analysis for assessing the risks of vulnerabilities exploitation and analyzing the impact of countermeasures on such risks.

Keywords Security requirements engineering · Risk analysis · Agent-oriented software engineering · Empirical security knowledge

1 Introduction

Developing secure software systems is challenging, because errors and misspecifications in requirements, design, and implementation can bring vulnerabilities to the system. Attackers most often exploit vulnerabilities to compromise the system. In security engineering, a vulnerability is an error or weakness of the IT system or its environment that in conjunction with an internal or external threat can lead to a security failure [1]. For example, vulnerabilities may result from input validation errors, memory safety violations, weak passwords, viruses, or other malware.

In recent years, software companies and government agencies have become particularly aware of security risks that vulnerabilities impose on the system security and have started analyzing and reporting detected vulnerabilities of products and services. For instance, the IBM Internet Security Systems X-Force [2] has detected and analyzed 6,437 new vulnerabilities in 2007, of which 1.9% are critical and 37% are high risk. Twenty percentage of the five-top critical vulnerabilities were found to be unpatched. Of all the vulnerabilities disclosed in 2007, only 50% can be corrected through vendor patches, and 90% of vulnerabilities could be remotely exploited. These statistics show the critical urgency of the vulnerabilities affecting software services and products. Various web portals and online databases of vulnerabilities are also made available to security administrators. For example, the National Vulnerability Database [3] SANS top-20 annual security risks [4], and Common Weakness Enumeration (CWE) [5] provide updated lists of vulnerabilities and weaknesses. The Common Vulnerability Scoring System (CVSS) [6] also provides a method for evaluating the criticality of vulnerabilities.

Existing software engineering frameworks focus on various aspects for eliciting security requirements such as design of secure components [7], security issues in social dependencies among actors [8] and their trust relationships [9], attacker behavior [10, 11] and attacker goals [12], and events that can cause system failure [13]. However, they rarely use vulnerabilities to elicit security requirements.

G. Elahi (✉) · E. Yu · N. Zannone
University of Toronto, Toronto, ON, Canada
e-mail: gelahi@cs.toronto.edu

E. Yu
e-mail: yu@ischool.utoronto.ca

N. Zannone
e-mail: zannone@cs.toronto.edu

Liu et al. [8] propose a vulnerability analysis approach for eliciting security requirements. However, vulnerabilities in this framework are different from the ones defined in security engineering (i.e., weaknesses in the IT system). Liu et al. refer to vulnerabilities as the weak dependencies that may jeopardize the goals of depender actors. Only few security software engineering approaches consider analyzing vulnerabilities, as weaknesses in the systems, during the elicitation of security requirements. For instance, in [14], vulnerabilities are modeled as beliefs inside the boundary of attackers that may positively contribute to attacks. However, the resulting models do not specify which actions or assets introduce vulnerabilities into the system and which actors are vulnerable. In addition, the impact of countermeasures on vulnerabilities and attacks is not captured. The CORAS framework [15, 16] provides a way for expressing how a vulnerability leads to another vulnerability and how a vulnerability (or combination of vulnerabilities) lead to a threat. However, similar to [14], CORAS does not investigate which design choice, requirement, or process has brought the vulnerabilities to the system.

Current state of the art raises the need for a systematic way to link the empirical security knowledge such as information about vulnerabilities, attacks, and proper countermeasure to stakeholders' goals and security requirements. By identifying vulnerabilities or classes of vulnerabilities and associating them with the activities and assets that bring them to the system, analysts can understand how weaknesses are brought to the system and how flaws in one part of the system are spread out to other parts. Information about potential attacks that exploit vulnerabilities can be linked to requirements to analyze the effects of the exploited vulnerabilities on activities or goals of stakeholders. Analyzing the effects of vulnerabilities on the system makes it possible to assess the risks of attacks, analyze the efficacy of countermeasures, and decide on patching or disregarding the vulnerabilities. In our previous work [17], we have introduced vulnerabilities into a security conceptual modeling method to address these issues. Vulnerabilities are treated as weaknesses in the structure of goals and activities of intentional agents. However, vulnerabilities were only graphically attached to the i* modeling elements, while the semantics of vulnerabilities relationships with other elements of the i* models were not well defined. In addition, vulnerability analysis was not complemented with a threat analysis.

This paper extends and refines our previous work by proposing an agent- and goal-oriented framework for eliciting and analyzing security requirements by linking empirical knowledge of vulnerabilities to requirements models. In particular, we have revised the modeling framework proposed in [17] on the basis of a conceptual framework centered on vulnerabilities. This conceptual framework helps us identify the basic constructs and relationships necessary to model and analyze vulnerabilities and their effects on the system, and define their semantics. The proposed vulnerability-centric security requirements framework is the result of surveying current critical vulnerabilities in security engineering discipline to understand how vulnerabilities are brought to the system, exploited by the attacks, and handled by the countermeasures.

Together with a modeling framework, this paper proposes a goal model evaluation method that helps analysts verify whether top goals of stakeholders are satisfied with the risk of vulnerabilities and attacks and assess the efficacy of security countermeasures against such risks. The evaluation does not only specify if the goals are satisfied, but also makes it possible to understand why and how the goals are satisfied (or denied) by tracing back the evaluation to vulnerabilities, attacks, and countermeasures. In addition, the resulting security goal models and goal model evaluation can provide a basis for trade-off analysis among security and other quality requirements [17]. New vulnerabilities are continuously being uncovered. By linking requirements, vulnerabilities, and countermeasures to each other in a modeling framework, one can update the models with newly detected vulnerabilities in order to analyze the risks imposed by the new vulnerabilities.

The structure of the paper is organized as follows. Section 2 introduces the security concepts used in the paper with a particular focus on vulnerabilities and related notions. Section 3 introduces the meta-model of the framework, in which security concepts are incorporated into an agent- and goal-oriented modeling framework. Section 4 describes the modeling process, and Sect. 5 proposes a method for analyzing security requirements based on the goal model evaluation techniques. The modeling and analysis methods described are illustrated by case examples. Section 6 overviews the current state of the art in threat analysis and security requirements engineering. Finally, Sect. 7 draws a conclusion and discusses future work.

2 Relevant concepts

This section investigates the conceptual foundation for the security requirements engineering framework proposed in this paper. We identify and discuss the basic security conceptual modeling constructs that we have adopted in the meta-model of our framework (Sect. 3). This discussion is grounded in the security engineering literature.

An *asset* is *"anything that has value to the organization"* [18]. Assets can be people, information, software, and hardware [16]. They can be the target of attackers and, consequently, need to be protected. Assets such as software products, services, and data may have vulnerabilities. A *vulnerability* is a weakness or a backdoor in the IT system which allows an attacker to compromise its correct behavior [1, 19, 20]. Identifying which are the vulnerabilities of the system and which assets have brought them into the system help analysts to analyze how vulnerabilities spread within the system and, consequently, to determinate the vulnerable components of the system.

The potential way an attacker can violate the security of (a component of) the IT system is called *threat* (or *attack*) [21]. Essentially, an *attack* is a set of intentional unwarranted *actions* which attempts to compromise confidentiality, integrity, availability, or any other desired feature of an IT system. Though the general idea of attack is clear, there is no consensus on a precise definition. For instance, Schneider [20] points out that an attack can occur only in presence of a vulnerability. Conversely, Schneier [21] broadens this vision, considering also attacks that can be performed without exploiting vulnerabilities. Several frameworks for security analysis take advantage of *temporally ordered* models for analyzing attacks [22, 23]. Incorporating the concept of time into the attack modeling helps to understand the steps of actions and vulnerability exploitations which lead to a successful attack. However, temporally ordered models add complexity to the requirements engineering models which may not be suitable for the early stages of development.

Analyzing attacks and vulnerabilities allows analysts to understand how system security can be compromised. Another aspect to be considered is attackers' motivations (*malicious goals*). Examples of malicious goals are *disrupt or halt services*, *access confidential information*, and *improperly modify the system* [24]. Schneier [10] argues that understanding who the attackers are along with their motivations, goals, and targets, aids designers in adopting proper countermeasures to deal with the real threats. Analyzing the source of attacks helps to better predict the actions taken by the attacker.

Threat analysis attempts to identify the types of threats that an organization might be exposed to and the harm they could cause to the organization (i.e., the *severity* of threats). Threat analysis starts with the identification of possible attackers, evaluates their goals, and how they might achieve them. Through threat assessment, analysts can assess the risk and cost of attacks, and understand their impact on system security.

Such knowledge helps analysts in the identification of appropriate countermeasures to protect the system. A *countermeasure* is a protection mechanism employed to secure the system [21]. Countermeasures can be actions, processes, devices, solutions, or systems intended to prevent a threat from compromising the system. For instance, they are used to patch vulnerabilities or prevent their exploitation.

Besides the concepts described earlier, there are other concepts relevant to security requirements. For instance, Massacci et al. [25] integrate concepts from trust management, such as permission, trust, and delegation, into a requirements engineering framework to address authorization issues in the early phases of software development process. Risk analysis frameworks (e.g., [13]) employ the concepts of event to model uncertain circumstances that affect the correct behavior of the system. However, events do not allow the analysis of (malicious) intentional behavior and, therefore, they result more appropriate to assess risks and elicit safety requirements in critical systems.

Security is not only limited to the identification of protection mechanisms to address vulnerabilities. Security originates from human concerns and intents [8]; the social issues of organizations where different actors can collaborate or compete to achieve their goals should be considered as part of security requirements analysis [8, 9]. In addition, security is a subjective and personal feeling [26]; therefore, security requirements analysis and security-related decision makings require analyzing personal and organizational goals of the stakeholders participating to the system. For this purpose, we take advantage of agent- and goal-oriented concepts such as intentional actor, goal, and social dependency. There is evidence in the security requirements engineering literature (e.g., [8, 9, 12, 27]) that these concepts provide the means for analysis of organizational and social contexts in which the system-to-be operates. In the next section, we show how security concepts can be integrated in the meta-model underlying the i* agent- and goal-oriented framework.

3 An extended i* meta-model

Security is both a technical and a social/organizational problem. The ability of the i* framework [28] to model agents, goals, and their dependencies makes it suitable for understanding security issues that arise among multiple malicious or non-malicious agents with competing goals. In addition to modeling actors, i* offers a way to model actors' dependencies, goals, assets, and actions, refinements of goals into the actions and assets, and decomposition of actions. Thus, the i* framework provides the basic setting for representing vulnerabilities that are brought by actions and assets and propagating them through the decomposition and dependency links to other elements of model. Moreover, i* enables modeling contribution of goals, actions, and assets on other goals. Such relations can be used to capture the effects of vulnerabilities on the satisfaction of system and stakeholders' goals.

In this section, we present the meta-model for the security requirements engineering framework, which extends the i* meta-model with security concepts (Fig. 1). The meta-model includes both the i* strategic dependency (SD) diagram, which captures the actors and their dependencies and the i* strategic rationale (SR) diagram, which expresses the internal goals and the behavior of actors to achieve their goals. The meta-model also captures the concepts of vulnerability, attack, security countermeasure, and their corresponding relationships with i* constructs.

3.1 The i* meta-model

This section provides an overview of the i* framework's meta-model along the modeling constructs it provides (Fig. 1). An actor is an active entity that has strategic goals and intentionality within the system or the organizational setting, carries out activities, and produces entities to achieve goals by exercising its knowhow [28]. Actors can be roles or agents. A *role* captures an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor. An *agent* is an actor with concrete and physical manifestations and can play some role.

Intentional elements defined by the i* framework are goals, softgoals, tasks, and resources. A *goal* represents the intentional desire of an actor, without specification of how the goal is satisfied. Goals are also called hard goals in contrast to *softgoals* which do not have clear criteria for deciding whether they are satisfied or not. A *task* is a set of actions which the actor needs to perform to achieve a goal. A *resource* is a physical or an informational entity and is used to represent assets.

The relations between actors are captured by the notion of *dependency*. Actors can depend on each other to achieve a goal, perform a task, or furnish a resource. For example, in a goal dependency, an actor (the *depender*) depends on another actor (the *dependee*) to satisfy the goal (the *dependum*). In addition to the dependum, two other intentional elements are involved in a dependency. One element represents *why* a depender needs the dependum, and the other element specifies *how* the dependee satisfies the dependum.

The meta-model in Fig. 1 also describes the relationships between intentional elements inside the *boundary* of actors. Actors have (soft)goals and rely on other (soft)-goals, tasks, and resources to achieve them. Softgoals can be decomposed into more softgoals using *AND/OR decomposition* relations. *Means-end* links are relations between goals and tasks, and indicate that a goal (the *end*) can be achieved by performing alternative tasks (the *means*). Tasks can be decomposed into any other intentional elements through *task decomposition* links. By decomposing a task into subelements, one can express that the subelements need to be satisfied or available to have the root task performed.

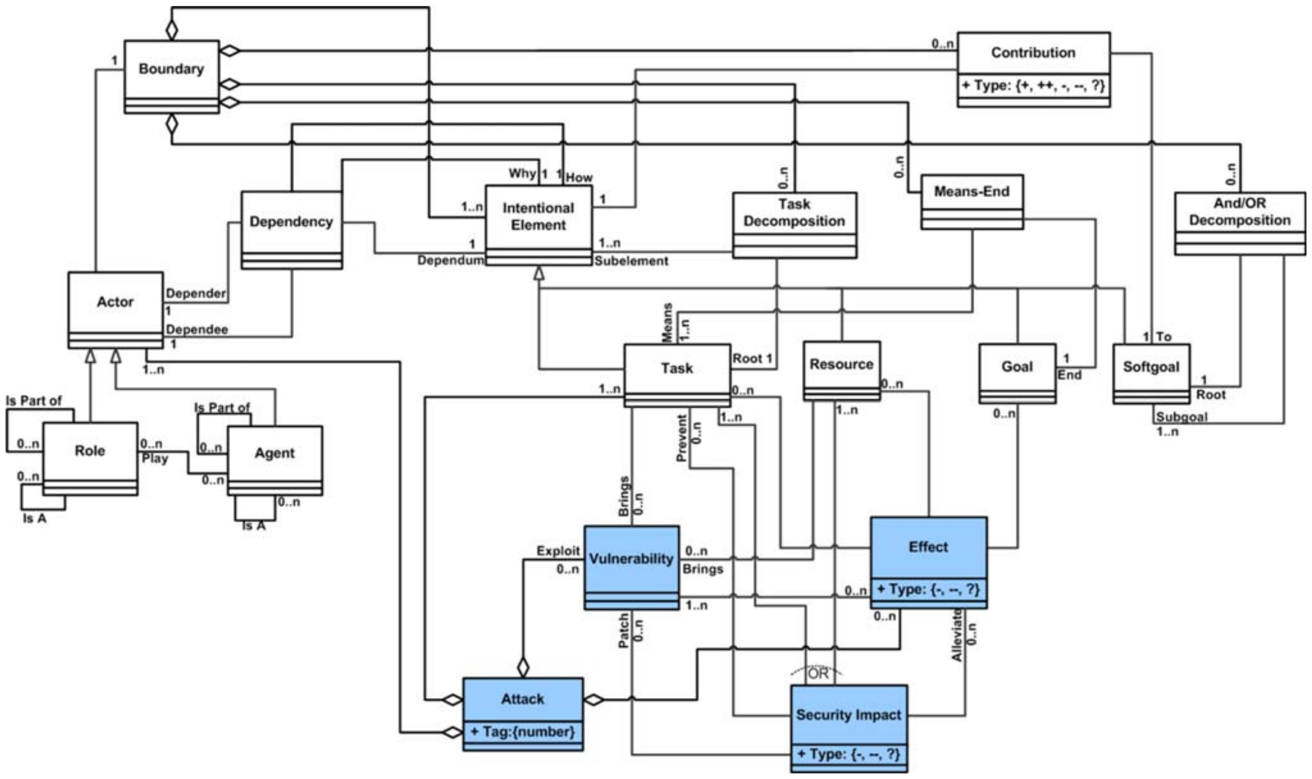

Fig. 1 The i* meta-model extended with the concept of vulnerability and attack

Softgoals and other intentional elements can contribute either positively or negatively to the other softgoals. This is expressed by the *contribution* links. The contribution relation is characterized by an attribute *type* which can be Help (+), Make (++), Hurt (−), Break (−−), and Unknown (?) values. By linking an intentional element to a softgoal by a Make (Break) contribution, one can express that satisfaction of the intentional element is enough to fully satisfy (fully deny) the softgoal, while Help (Hurt) contributions indicate that the intentional element has positive (negative) impact, but the impact is not enough to fully satisfy (deny) the softgoal. This qualitative approach for modeling contribution to softgoals reflects the fact that softgoals do not have clear-cut satisfaction criteria.

3.2 Attack and security countermeasure extensions to the i* meta-model

The concepts of vulnerability, attacks, effects of vulnerabilities, and impact countermeasures are added to the i* meta-model. In Fig. 1, extended elements to the i* meta-model are highlighted. Adopting a task or employing a resource can bring *vulnerabilities* to the system. The concept of vulnerability is not limited to specific reported vulnerabilities or to general classes of vulnerabilities. For example, one can model the famous worm called *2000 ILOVEYOU*[1] or general class of *argument injection or modification*. For the sake of simplicity, we call an intentional element that introduce a vulnerability a *vulnerable element*. Vulnerabilities are concrete weaknesses or flaws that exist in a component of the system like a process, asset, algorithm, and platform, whereas goals and softgoals represent actors' intentions and software quality attributes, respectively. In the i* conceptual framework, adopting a task or employing a resource describes a concrete way of achieving a (soft)goal; therefore, (soft)goals which are abstract and independent of operationalization do not introduce a flaw or vulnerability.

Exploitation of vulnerabilities can have an *effect* on the same element that has brought the vulnerabilities or on other tasks, goals, and resources. The effect is characterized by an attribute, *type*, which specifies how the vulnerability affects a goal, a task, or a resource. The effect types are Hurt (−), Break (−−), and Unknown (?). The effect of vulnerabilities on softgoals is not considered in the meta-model, since softgoals are not directly measurable goals. The effect of vulnerabilities would be propagated to softgoals from the affected elements that contribute to the softgoals.

An *attack* represents the set of actions that an attacker performs to *exploit* a number of vulnerabilities and has negative effects on other intentional elements. In Fig. 1, we use an aggregation relation to indicate the tasks, actors, vulnerabilities, and their effects are assembled and configured together to mount an attack. This definition of attacks is based on the definition proposed by Schneider [20] in which vulnerabilities are a key aspect of any attack. This choice is due to the fact that we are mainly interested in analyzing the effects of vulnerabilities on the system. Attacks that are performed without exploiting vulnerabilities can be modeled by introducing a new class of attacks in which their target is a task or a resource instead of a set of vulnerabilities.

Resources and tasks can have a *security impact* on attacks. Such tasks and resources can be interpreted as security countermeasures; however, we do not distinguish them from non-security mechanisms in the meta-model as this distinction does not affect the requirements analysis. The impact relation has the attribute, *type*, which accepts Hurt (−), Break (−−), and Unknown (?) values. Security countermeasures can be used to *patch* vulnerabilities, *alleviate* the effect of vulnerabilities, or *prevent* the malicious tasks that exploit vulnerabilities or system functionalities.

By patching the vulnerability, the countermeasure fixes the weakness in the system. Example of such countermeasure is new updates the software vendors provide for the released products. A countermeasure that alleviates the vulnerability effects does not address the source of the problem, but it intends to reduce the effects of the vulnerability exploitation. For example, a backup system mitigates the effect of security failures that cause data loss. Countermeasures can prevent the actions that the attacker performs, which consequently prevents exploitation of the vulnerability by the actions. For example, an authentication solution prevents unauthorized access to assets. Countermeasure may prevent performing vulnerable tasks or prevent using vulnerable resources, which results in removing the vulnerability that has been brought to the system by the vulnerable elements. For example, one can disable JavaScript option in the browser to prevent exploitation of malware run by the browser.

The definition of attack and security countermeasure is fundamentally a matter of perspective: a task or a goal counted as malicious can be perceived non-malicious from a different viewpoint. Sequences of actions for mounting an attack are basically similar to sequences of actions performed by legitimate actor. Therefore, the line to differentiate malicious actions from non-malicious ones is arbitrary, and distinguishing malicious goals from non-malicious goals depends on the perspective adopted by the system designer.

Malicious elements have the same semantics of ordinary intentional elements: they can be similar or identical to

[1] http://www.cert.org/advisories/CA-2000-04.html.

non-malicious elements. For example, the desire to have a high profit is not a malicious goal, but an actor can achieve such a goal either by working legally and honestly or cheating. On the other hand, a task can be interpreted as malicious in a condition, while it is counted as non-malicious in a different context. For example, one can install a camera for spying into other people privacy, whereas a surveillance camera can be used for security purposes. In this example, the goal for performing tasks indicates whether the task is malicious or not.

Since malice is a matter of perspective, distinguishing malicious and non-malicious behavior does not affect security requirements analysis. Therefore, the meta-model presented in Fig. 1 is a neutral meta-model that does not distinguish malicious and non-malicious elements. However, as showed by Sindre and Opdahl [29], graphical models become much clearer if the distinction between malicious and non-malicious elements is made explicit and the malicious actions are visually distinguished from the legitimate ones. Sindre and Opdahl show that the use of inverted elements strongly draws the attention to dependability aspects early on for those who discuss the models. In this regard, an extended meta-model is developed with the assumption that some actors are attackers and have malicious goals, and other actors employ countermeasures for protecting their goals. Figure 2 presents the extended meta-model, which is derived from the meta-model in Fig. 1 by introducing a new type of actor called *attacker*. An attacker is a specialization of the i* actor elements; thus, the same modeling rules and properties of the i* actors can be applied for modeling attackers. In particular, as for actors, attackers can be roles and agents. Attackers have malicious intentional elements such as *malicious goals* and *malicious softgoals* inside its boundary. The concept of boundary is added to link the malicious elements to the attacker. An *attack* involves an *attacker*, *malicious tasks* that he performs to exploit a set of *vulnerabilities*, and the *effect* of exploited vulnerabilities on other actors' intentional elements.

4 The modeling process

This section presents the security requirements modeling process along with the modeling notation and graphical representation. The resulting models help analysts to

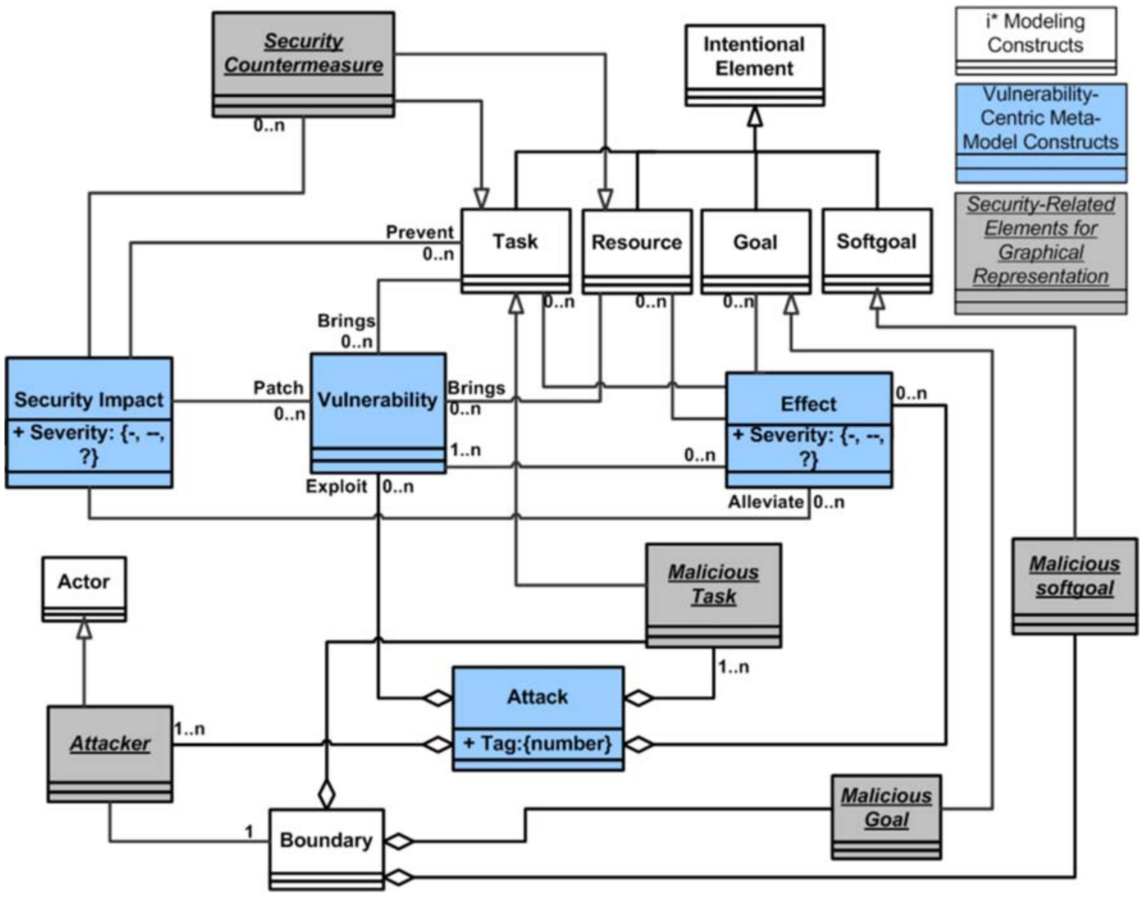

Fig. 2 A fragment of the meta-model in which attacks, attackers, malicious behavior, and security countermeasure are explicit elements of the meta-model

understand the social and organizational dependencies among main stakeholders of the system, their goals, the system architecture, the organization structure [28], and security issues that arise among interaction of actors [17] in the early stages of the development.

Figure 3 summarizes the modeling process. The process consists of five steps; each of them results in a *view* of the security requirements model. Each of these views provides additional incremental information:

1. *Requirements view* captures stakeholders and system actors together with their (soft)goals, the tasks to achieve those goals, required resources, and the dependencies among them.
2. *Vulnerabilities view* extends the requirements view by adding the vulnerabilities that tasks and resources brings to the system and the impact that their exploitation (or of their combinations) has on the system.
3. *Attackers template view* captures the behavior of attackers by representing how attackers can exploit vulnerabilities to compromise the system.
4. *Attackers profile view* captures individual goals, skills, and behavior of a specific class of attackers based on the attacker template view.
5. *Countermeasures view* captures the security solutions adopted by actors to protect the system and their impacts on attacks and vulnerabilities.

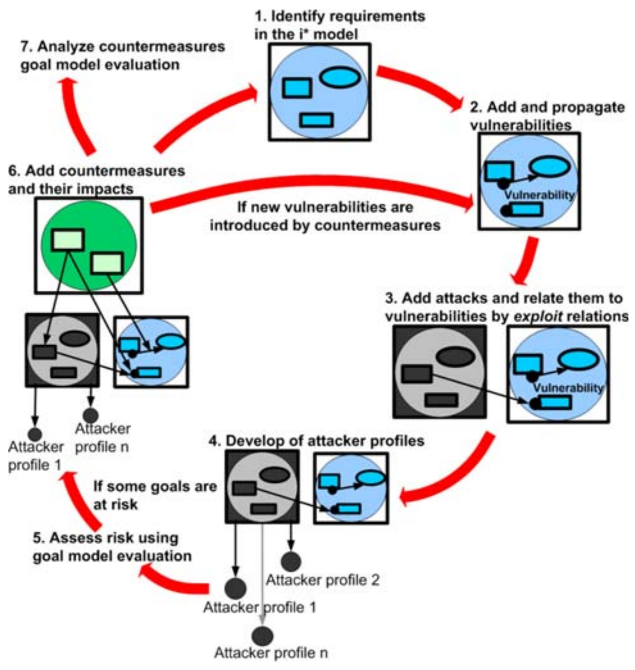
1. Identify requirements in the i* model
2. Add and propagate vulnerabilities
Vulnerability
3. Add attacks and relate them to vulnerabilities by *exploit* relations
Vulnerability
4. Develop of attacker profiles
Attacker profile 2
Attacker profile 1
Attacker profile n
5. Assess risk using goal model evaluation
If some goals are at risk
6. Add countermeasures and their impacts
If new vulnerabilities are introduced by countermeasures
Attacker profile 1
Attacker profile n
7. Analyze countermeasures goal model evaluation

Fig. 3 The modeling process

The process for developing security requirements models is incremental: in each step, new elements are added to the requirements model to show new aspects. The modeling process starts with the identification of actors, their dependencies, goals, and the tasks and resources necessary to achieve them. Then, vulnerabilities are identified and propagated through the goal model. In the third step, possible attacks that can exploit the vulnerabilities are identified and analyzed. Attacker profiles that specify the capabilities and skills of categories of attackers are defined. The model is then evaluated to assess the risks of exploitation of vulnerabilities by attackers. If the analysis shows that the risks cannot be tolerated by stakeholders, the requirements model is revised by introducing countermeasures and their impacts on vulnerabilities and attacks. Modeling goal, vulnerabilities, attacks, and countermeasures is an iterative process as the adoption of countermeasures may cause the introduction of new vulnerabilities as well as denial of functionalities or quality goals.

Identification of vulnerabilities, attacks, and countermeasures requires security knowledge and experience. A main assumption of this work is that analysts have the security experience and knowledge necessary to identify vulnerabilities or extract them from vulnerabilities knowledge bases. The proposed framework does not provide guidelines or methods for finding vulnerabilities and attacks and identifying proper countermeasures. It proposes a way for linking security knowledge such as reports of attacks, list of vulnerabilities, and alternative countermeasures, to requirements and provide support for security requirements analysis. Analysts can take advantage of available vulnerability knowledge sources such as the CWE categorization of weaknesses and vulnerabilities [5], SANS list of top-20 vulnerabilities [4], and CVE entries [6]. CVE contains vendor-, platform-, and product-specific vulnerabilities. Such technology- and system-oriented vulnerabilities are not useful to decide on security requirements in the early stages of development where target platform and technology is not yet decided. SANS list and CWE catalog include more abstract weaknesses, errors, and vulnerabilities. Some entries in these lists are technology and platform independent, while others are specific to specific products, platforms, and programming languages.

4.1 Eliciting and modeling initial requirements

Requirements modeling intends to identify and model stakeholders' needs and system requirements. We take advantage of the i* framework that provides a way for modeling and analyzing stakeholders' and system's goals and system-and-environment alternatives that address achievements of the goals [28, 30]. We do not present the modeling process underlying the i* framework, and details in this regard can be found in [30].

Figure 4 shows the requirements view of a browser which requests the content from a web server to build HTML pages. The *User* depends on a software agent, the *Firefox Browser*, to *Browse Web sites*. The browser depends on the *User* to *Enter inputs* and depends on *Web server* for *Web page dynamic content and JavaScript*. To get the dynamic content, the browser needs to get content which is entered by other users (*User's inputs*). This view also describes high-level goals and tasks of the browser. For instance, one of the Firefox browser's tasks is to *Show the web pages* and to perform that the browser needs to *Run the JavaScript with user inputs*. This makes the final customized HTML page, and for this aim, the browser *Request and get pages from the server* and *Get users' input*.

4.2 Modeling and analyzing vulnerabilities

Vulnerabilities modeling intends to understand the weaknesses affecting system requirements. To incorporate specific vulnerabilities or classes of vulnerabilities into the requirements model, we incrementally refine the requirements view by identifying the vulnerable tasks and resources and analyze the effect of vulnerability exploitation. To represent vulnerabilities, the i* modeling notation is enriched with a black circle for the new graphical element. The black circle is chosen to resemble a hole or weakness in the system which leaves a backdoor for attacks. Vulnerabilities are graphically attached to tasks and resources, which implies the execution of the task or availability of the resource brings the vulnerability to system. To represent the possible effect(s) of an exploited vulnerability on goals, tasks, and resources, a new link is added to the i* relations. The vulnerability effect is visually represented by a dotted line with a label, *l*, where $l \in \{-, --, ?\}$.

Figure 4 shows the vulnerability view for a browser which requests the content from a web server. The model does not cover all possible vulnerabilities, and only shows some examples of vulnerabilities affecting web servers and browsers. One of the browser's tasks is to *Show the web pages*, and to perform that, the browser needs to *Run the JavaScript with user inputs*. The browser *Request and get pages from the server* and *Get users' input*. Each of these tasks bring a vulnerability to the system. By downloading a JavaScript code from the web server, a *Malicious Script* can be downloaded as well. The user inputs can also contain *Malicious input*. As a result, when the browser runs the JavaScript with the user inputs, the browser is exposed to the combination of the *Malicious script* and *Malicious user input* vulnerabilities.

When an actor depends on another actor for a vulnerable task or resource, the vulnerability is carried to the depender

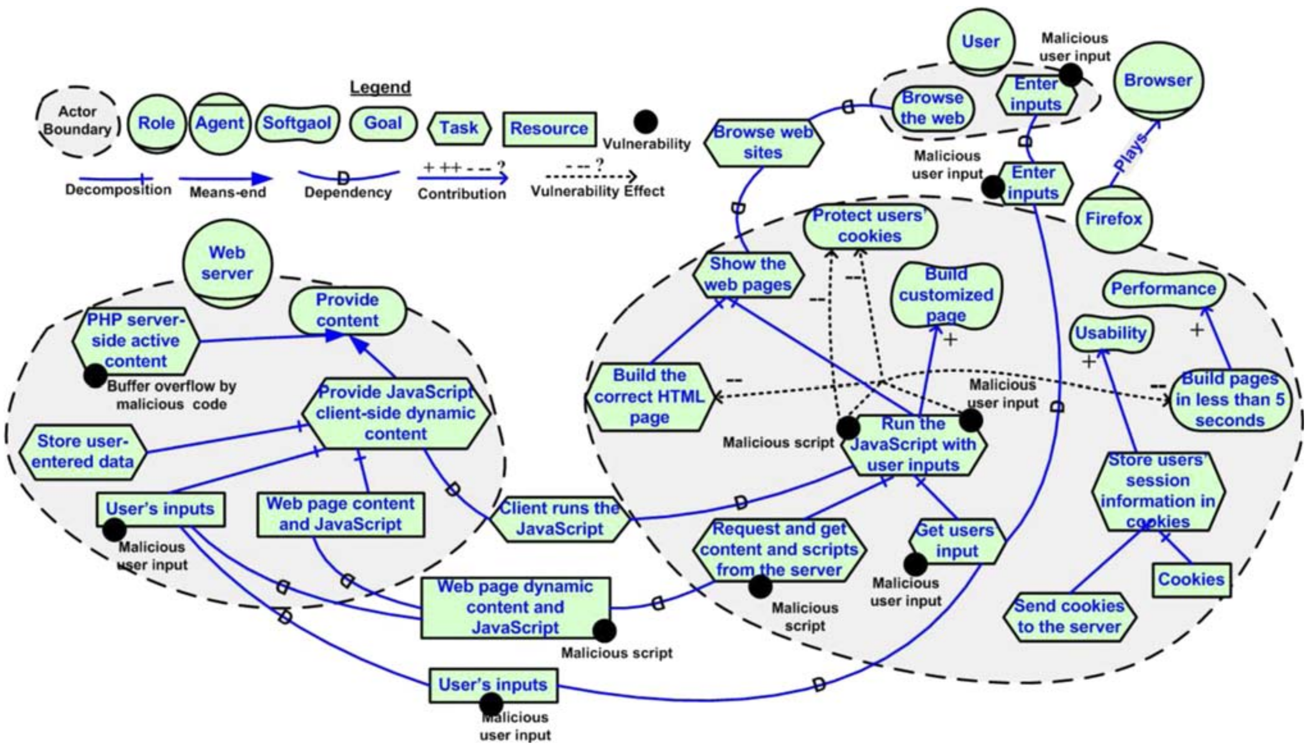

Fig. 4 Initial requirements and actors' actions, extended with the vulnerabilities view

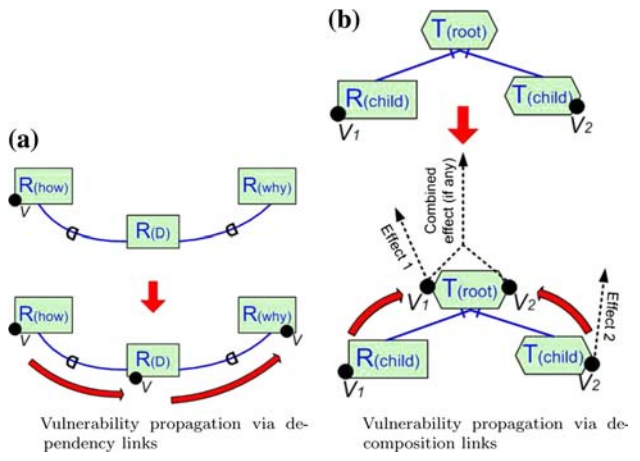
(a)
(b)
R(how)
R(why)
R(D)
V
T(root)
R(child)
T(child)
V1
V2
Effect 1
Combined effect (if any)
Effect 2
Vulnerability propagation via dependency links
Vulnerability propagation via decomposition links

Fig. 5 Vulnerability propagation rules

actor by the vulnerable dependum. Figure 5a explains the propagation of the vulnerability in the reverse direction of dependencies. This figure shows that for dependency relations, the vulnerability V in the dependee's resource $R_{(\text{how})}$ is propagated to the dependum, R_D, and the depender's element, $R_{(\text{why})}$. For example, in Fig. 4, the *Web server* depends on the users for *User's inputs* and store the users' provided content. Example of such web servers are Wiki pages and discussion rooms that store users' provided content. Later, other users depend on the *Web server* for *Web page dynamic content and JavaScript*. However, the *User's input* vulnerability gets propagated to other users as the *Malicious Script* vulnerability. The *Malicious Script* is brought to the *Firefox* agent because of the dependency link between the browser and *Web server*. Similar arguments also applies to task dependencies.

Vulnerabilities are also propagated through *decomposition links*. Using decomposition links, analysts refine tasks into more detailed elements with higher resolution information, and the subelements describe the up level task in detail. The application of a framework to a number of case studies has shown that it is easier to identify vulnerabilities for concrete subelements rather than for high-level abstract ones. Therefore, vulnerabilities are propagated bottom-up from subelements to the high level decomposed task. Figure 5b depicts the vulnerability propagation rule through decomposition links. This figure depicts that if a task T_{root} is decomposed into a task T_{child} and a resource R_{child}, respectfully with vulnerabilities V_1 and V_2, the root task would receive both vulnerabilities V_1 and V_2. Vulnerability effects depend on the context of the vulnerable elements. As shown in Fig. 5(b), the analyst can either assign the vulnerability effect to the child (*Effect*$_2$ for V_2) or to the root (*Effect*$_1$ for V_1) element based on the context. In addition, based on the context, one may determine that the propagated vulnerabilities have a combined effect. Propagating vulnerabilities effects cannot be automatically deducted from the structure of the model and requires human judgment and security experiences.

A concrete example of vulnerability propagation through decomposition links is shown in Fig. 4 where the *Run the JavaScript with user inputs* task in decomposed into *Request and get content and scripts from the server* with *Malicious script* vulnerability and *Get users' input* with *Malicious user input* vulnerability. Accordingly, the root task receives both vulnerabilities. These vulnerabilities or their combination can have various effects on goals and tasks of the actors when running the JavaScript. Figure 6 shows how the vulnerabilities are combined. The effect of exploiting the combination of *Malicious script* in the *malicious user input* vulnerabilities is expressed using the vulnerability effect link with a – – (*break*) contribution from the combination of the vulnerabilities to *Protect users' cookies* and *Build the correct HTML page*. In the next sections, we describe how the requirements and vulnerabilities views are related to the attacker template and countermeasure views.

4.3 Modeling attacker templates

The aims of attacker template modeling is to define a view of the security requirements model that represents the possible ways in which attackers can exploit vulnerabilities to compromise the system and the goals behind these attacks. To build the attacker template, designers can take advantages of existing approaches (e.g., attack tree [10] and anti-goals [12]) to develop a tree-like malicious goal model. In addition, catalogs of malicious goals [24] might be useful for driving attacker goals.

As discussed earlier, the proposed modeling notation graphically distinguishes malicious and non-malicious elements using a black shadow in the background of malicious elements as proposed in [8, 17]. The exploitation of a vulnerability by a malicious actor is graphically represented by a link labeled *exploit* from the malicious task to the vulnerability. A vulnerability may have different effects on other goals and mechanisms. Different attacks that exploit a vulnerability may have different effects on other elements. Therefore, to graphically link an attack and the effect of the vulnerability that the attack exploits, the corresponding *vulnerability effect* links for each attack are labeled with the same *tag number* that the exploit link is tagged. In this way, an attack is a quadruple consisting of an *attacker*, the *malicious task* that the attacker performs, a set of *vulnerabilities*, and their *effect* on the system (see Fig. 2).

Figure 6 extends a fragment of the requirements view in Fig. 4 by introducing two possible attackers: *Random hacker* and *Fake Web Site*. *Fake Web Site* is a *Web*

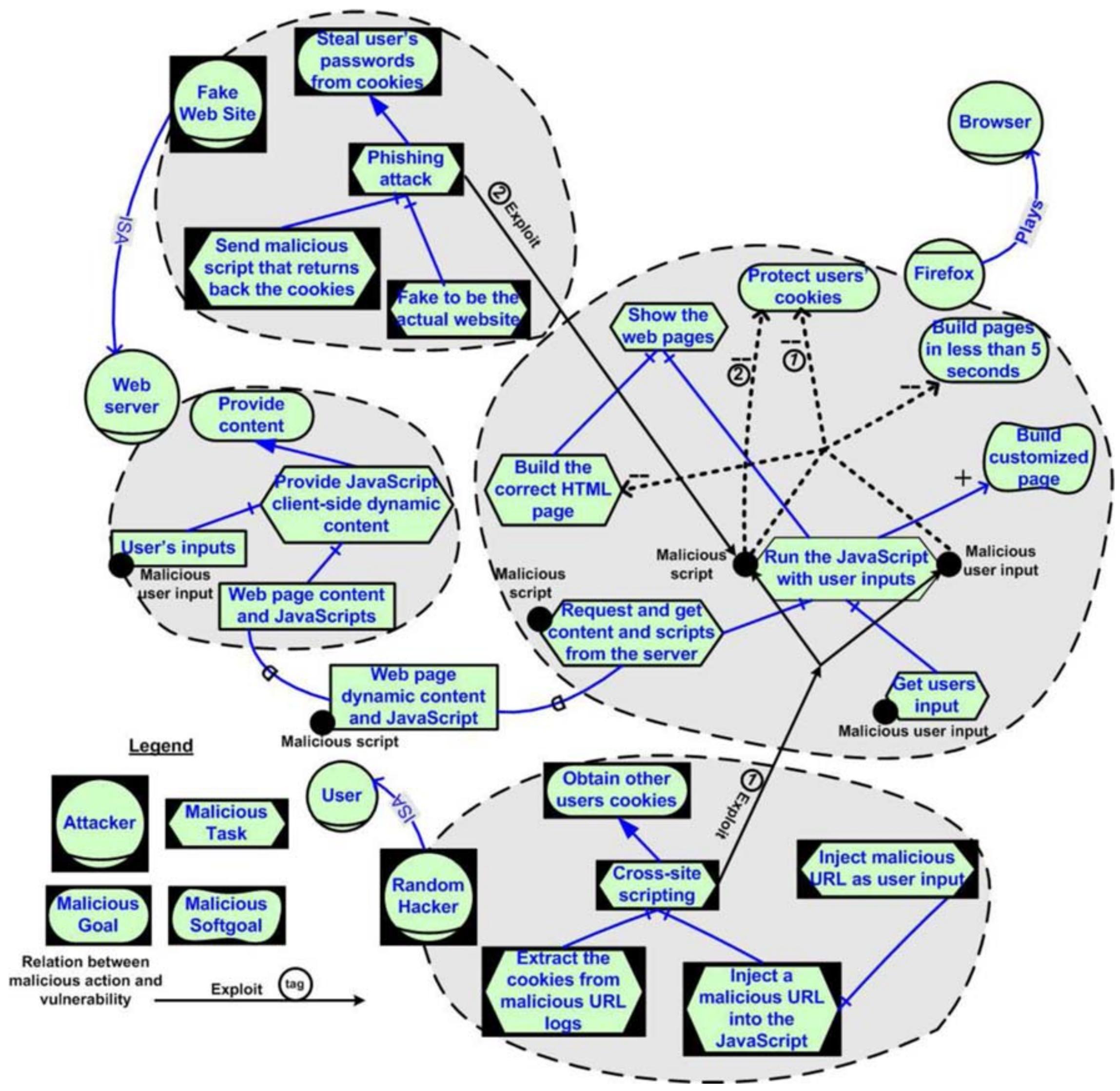
Steal user's passwords from cookies
Fake Web Site
Phishing attack
Send malicious script that returns back the cookies
Fake to be the actual website
ISA
Exploit
Browser
Plays
Firefox
Protect users' cookies
Show the web pages
Build pages in less than 5 seconds
Build customized page
Build the correct HTML page
Web server
Provide content
Provide JavaScript client-side dynamic content
User's inputs
Malicious user input
Web page content and JavaScripts
Malicious script
Run the JavaScript with user inputs
Request and get content and scripts from the server
Get users input
Malicious user input
Web page dynamic content and JavaScript
Legend
Attacker
Malicious Task
Malicious Goal
Malicious Softgoal
Relation between malicious action and vulnerability
Exploit
tag
User
Random Hacker
Obtain other users cookies
Cross-site scripting
Inject malicious URL as user input
Extract the cookies from malicious URL logs
Inject a malicious URL into the JavaScript

Fig. 6 Attacker template view for the browser and web server example

server who intends to *Steal user's passwords from cookies*. The *Fake Web Site* is modeled in Fig. 4 as an (inverted) i* role since we refer to a generic web site rather than to a specific web site. The *Fake Web Site* uses *Phishing attack* by exploiting the *Malicious script*. The *Random hacker* role is a (malicious) specialization of *User* role and inherits the user's capabilities. For instance, the *Random hacker* can browse a website and enter inputs. The hacker can use these capabilities for his malicious intents such as *Obtaining other users cookies*. One possible way to obtain cookies of other users is *Cross-site scripting* which consists of *Injecting a malicious URL into the JavaScript* and *Extracting the cookies from malicious URL logs*. To *Inject a malicious URL into the JavaScript*, the hacker *Injects malicious URL as user input* by playing the role of an ordinary user. As discussed earlier, to specify which malicious task exploits the vulnerability and causes this effect both the exploit and vulnerability effect links are labeled with a tag (number one).

4.4 Identifying and modeling countermeasures

By developing requirements, vulnerabilities, and attacker template views, analysts have the machinery necessary to evaluate the risks threatening the system. On the basis of risk assessment, analysts elicit and analyze the security countermeasures needed to protect the system. To model countermeasures, a modeling element is not added to the i* framework, since countermeasures share the same nature with other tasks and resources. Different countermeasures can have a different impact on attacks. A countermeasure can *alleviate* the effect of a vulnerability, *patch* it, or *prevent* malicious tasks or system's functionalities that bring the vulnerabilities. These impacts are modeled through alleviate, patch, and prevent links, respectively.

The model in Fig. 7 presents the countermeasures for the vulnerabilities and attacks views in Fig. 6. The countermeasure elements are highlighted using a different color.[2] In Fig. 7, the web server employs two security mechanisms: *validate user input* and *Remove HTML tags from use input*. By removing the HTML tags, the malicious code is removed from the user input. This impact is modeled through *prevent* relations between the countermeasures and the malicious task *Inject malicious URL as user input* with "-" label. By validating user input, the *Malicious user input* vulnerability is partially patched. At the browser side, one can *Disable JavaScript* and use *Anti Phishing tool bar*. Disabling JavaScript prevents performing *Run the JavaScript with user inputs*, hence, the vulnerable task is not performed any more. As a result, the vulnerabilities that are brought by running JavaScript do not exist any more.

4.5 Attacker profile definition

Different typologies of attackers may have different capabilities and skills. The idea underlying the attacker profile is to analyze classes of attackers and their behavior against the system. To define capabilities and skills of a class of attackers, the tasks that the attacker can perform, resources that can obtain, and goals that can satisfy are identified and labeled. Intuitively, labels represent the evidences that a goal has been satisfied, a task has been performed, or a resource is available. We refer to Sect. 5 for details about evaluation labels. Table 1 gives two different attacker profiles for *Random Hacker* and two profiles for *Fake Web Site* introduced in Fig. 4. The table indicates which attacker can achieve the tasks by assigning evaluation labels to malicious tasks defined in the attacker template.

5 Security requirements analysis using goal model evaluation

In addition to the benefits that analysts gain through the modeling process, goal models including vulnerabilities, attacks, and countermeasures provide a basis for security requirements analysis. The purpose of the evaluation is to assess the risks and determine the countermeasure necessary to protect the system. While traditional risk analysis methods assess risks by considering quantitative probability and severity of successful attacks [21], we propose analyzing risks by evaluating satisfaction or denial of goals of the system and stakeholders. For this purpose, we take advantage of qualitative goal model evaluation techniques. Although a quantitative risk assessment approaches can greatly simplify decision making and provide accurate final results, it can be difficult to apply due to lack of agreed metrics of vulnerabilities and accurate measures, specially in the early stages of development. On the other hand, qualitative evaluation answers questions with lower resolution information, represented in a qualitative spectrum.

Goal model evaluation is the procedure to verify if the actors' top level goals are satisfied by the choices that they have made and consists of propagating denial or satisfaction evidences (labels) through the goal model using a set of rules [31]. Horkoff [32] proposes an i* goal model evaluation method where denial or satisfaction labels are assigned to the leaf nodes and then are propagated through the goal model based on the type of the links between the elements. Evidences for satisfaction or denial are in the scale of values, which ranges from full satisfy (S), partial satisfy (PS), unknown (?), and conflict (C) to partial deny (PD) and full deny (D) which intended order of $S > PS > C > ? > PD > D$. In case of conflicts, human judgment is required to resolve the conflicts of contributions during the evaluation process.

In this work, we have adopted and adapted the goal evaluation method in [32] to support a security requirements analysis centered on vulnerabilities. The enhanced syntax and semantics of i* for modeling security constructs requires revising the evaluation methods accordingly. For example, employing countermeasures introduces direct contributions of tasks and resources to malicious and non-malicious tasks, vulnerabilities, and effects of vulnerabilities. A security solution may partially prevent a malicious task. For instance, increasing the buffer size can only partially addresses buffer overflow unless the buffer size is made infinite. This impact may cause the propagation of partial satisfaction and denial values to hard elements such as tasks and goals. In contrast, the algorithm in [32] assumes that intentional elements contribute only to softgaols. This implies that partial values are only assigned to softgoals. In the current work, we relax this assumption by considering partial values to be assigned also to goals, tasks, and resources. Partial values associated with those elements indicate that there is partial evidence of their satisfaction or denial. Labels are also associated with vulnerabilities to represent the evidences that they are exploited or not.

Figure 8 summarizes the security goal model evaluation steps. The evaluation starts with assigning labels to the leaf nodes of both malicious and non-malicious actors. In

[2] The highlighted color in the models does not bear any semantic significance and only intends to highlight the countermeasures in the figures.

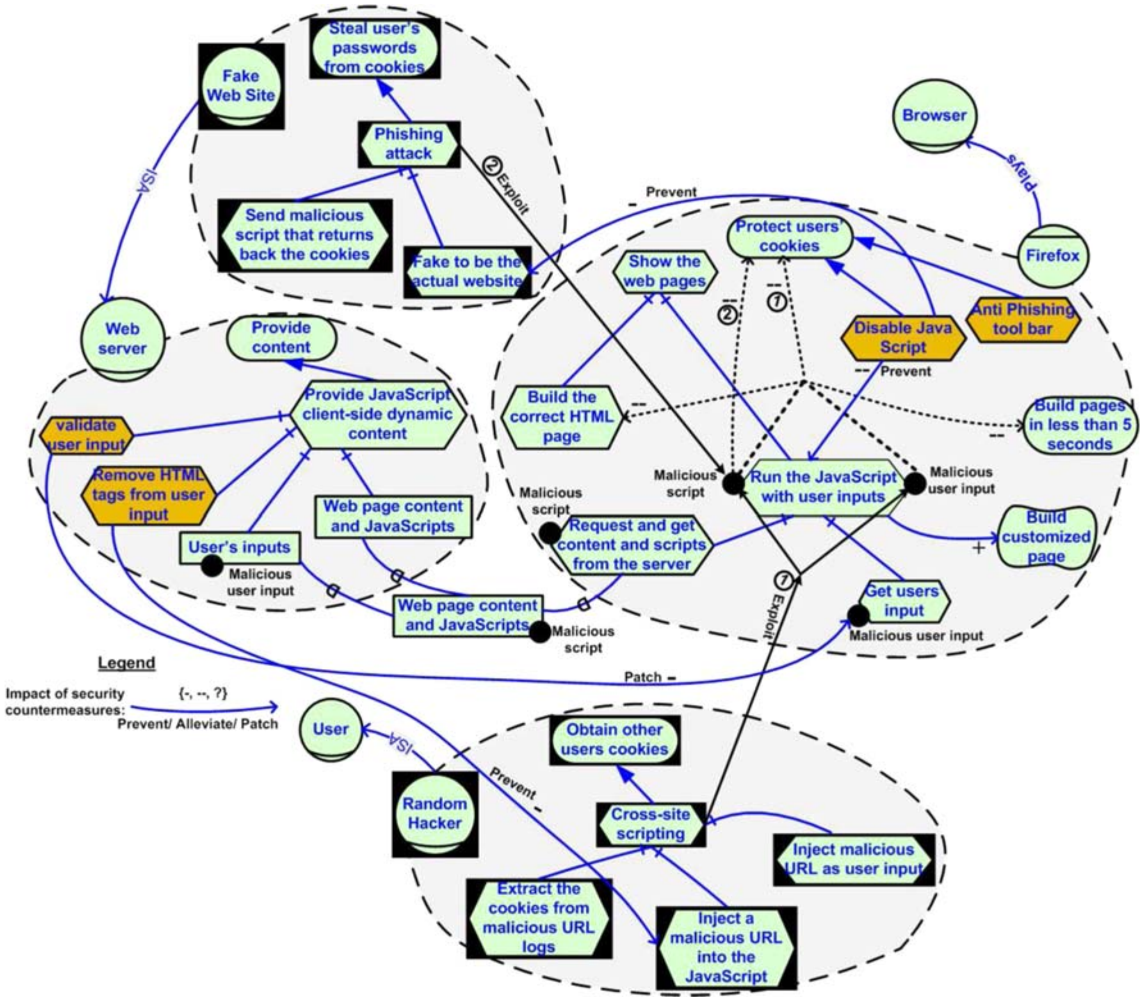
Steal user's passwords from cookies
Fake Web Site
Phishing attack
Send malicious script that returns back the cookies
Fake to be the actual website
ISA
Exploit
Browser
Plays
Prevent
Protect users' cookies
Show the web pages
Firefox
Anti Phishing tool bar
Disable Java Script
Prevent
Web server
Provide content
Provide JavaScript client-side dynamic content
validate user input
Remove HTML tags from user input
User's inputs
Malicious user input
Web page content and JavaScripts
Build the correct HTML page
Build pages in less than 5 seconds
Malicious script
Run the JavaScript with user inputs
Malicious user input
Request and get content and scripts from the server
Build customized page
Get users input
Malicious user input
Web page content and JavaScripts
Malicious script
Legend
Impact of security countermeasures:
{-, --, ?}
Prevent/ Alleviate/ Patch
Patch
User
ISA
Prevent
Obtain other users cookies
Random Hacker
Cross-site scripting
Inject malicious URL as user input
Extract the cookies from malicious URL logs
Inject a malicious URL into the JavaScript

Fig. 7 The countermeasure view for the web server and browser example. (The elements with highlighted color are countermeasures)

Table 1 Attacker profile definition for attacker templates in Fig. 4

Malicious task	Random hacker (1)	Random hacker (2)	Fake web site (1)	Fake web site (2)
Send malicious script that returns back the cookies	D	D	S	S
Fake to be the actual website	D	D	S	D
Inject a malicious URL into the JavaScript	S	S	D	D
Inject a malicious URL into the JavaScript	S	S	D	D
Extract the cookies from malicious URL logs	S	D	D	D

particular, the labels associated with malicious elements are defined based on the attackers' profile. Then, the labels are propagated to the upper nodes using propagation rules. The final result of evaluation shows the consequences of attacks and exploitation of vulnerabilities on higher goals. If some stakeholders and system goals are denied because of attacks, analysts need to consider countermeasures and analyze their impact on the system behavior and security. Evaluation labels are propagated through the goal model again to verify the satisfaction or denial of the goals because of the employment of countermeasures. In the remainder of this section, we present the propagation rules underlying the security goal evaluation method.

5.1 i* Propagation rules

This section briefly presents the propagation rules for i* concepts (i.e., dependency, contribution, decomposition, AND/OR, and means-end) based on the work in [32]. In a dependency relation, satisfaction of the depender element

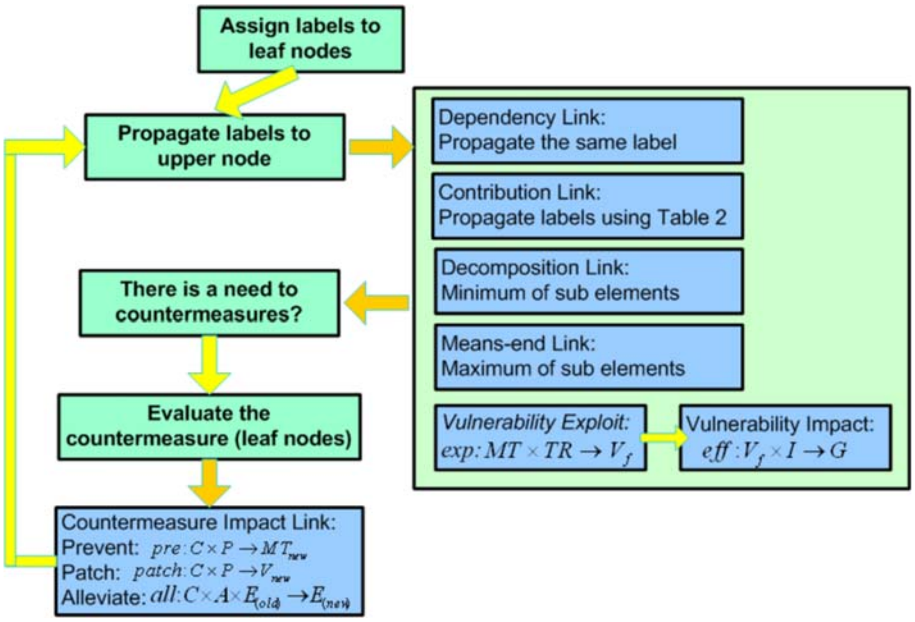
Assign labels to leaf nodes
Propagate labels to upper node
Dependency Link: Propagate the same label
Contribution Link: Propagate labels using Table 2
Decomposition Link: Minimum of sub elements
Means-end Link: Maximum of sub elements
Vulnerability Exploit: $exp: MT \times TR \rightarrow V_f$
Vulnerability Impact: $eff: V_f \times I \rightarrow G$
There is a need to countermeasures?
Evaluate the countermeasure (leaf nodes)
Countermeasure Impact Link:
Prevent: $pre: C \times P \rightarrow MT_{new}$
Patch: $patch: C \times P \rightarrow V_{new}$
Alleviate: $all: C \times A \times E_{(old)} \rightarrow E_{(new)}$

Fig. 8 Security goal model evaluation steps

relies on the satisfaction of the dependee element. Therefore, dependency links propagate the evaluation value of the dependee element to the dependum and then to the depender element. Differently from [32], we do not define rules for dealing with ambiguous scenarios. Analysts should avoid developing goal models that contains ambiguity such as the ones presented in the left side of Fig. 9. Analysts shall revise the models in order to disambiguate the model. The right side of Fig. 9 shows some examples of unambiguous scenarios.

Contribution links represent the impact of intentional elements only on softgoals. Table 2 presents the rules for propagating the impact of an intentional element with denial or satisfaction value on the target softgoal through contribution links. We relax the restriction related to contribution links in the i* syntax, so, a malicious or countermeasure task can have contribution to hard elements as well. The rules in Table 2 are also valid for contribution links to hard elements. However, the rules in Table 2 are not enough for propagating the evaluation values from multiple elements to a softgoal. Because one element can make the softgoal satisfied, another element may make the softgoal denied, which entails conflicting evidences. In this work, we follow the approach suggested in [32] where human judgment is required to resolve conflicts.

AND/OR links refine a softgoal into one or more softgoals. When a softgoal is refined using AND links, the *minimum* value among the sub softgoals is propagated to the higher softgoal. When the softgoal is refined using OR

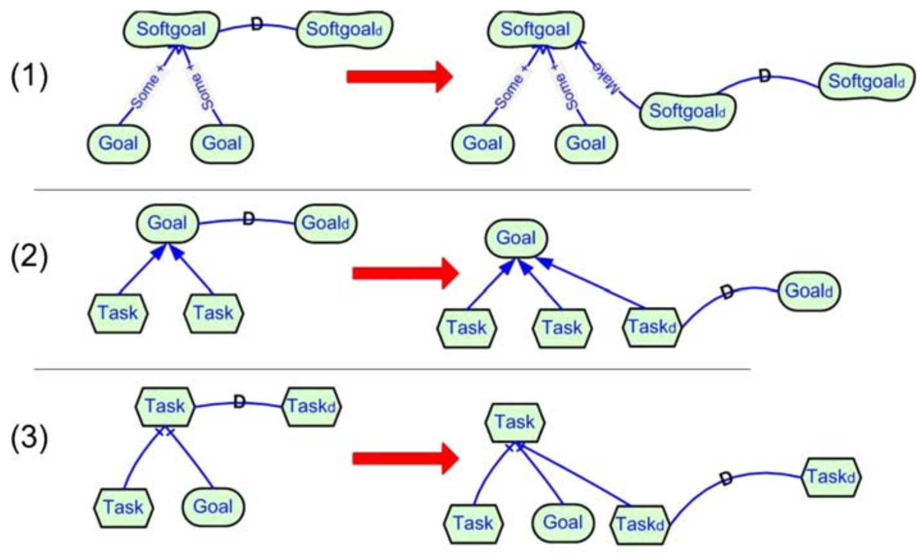

Fig. 9 Ambiguous dependency links which analysts need to avoid (Softgoal$_d$, Goal$_d$, and Task$_d$ are the dependum elements)

links, the *maximum* value among the sub softgoals is propagated to the top softgoal. A decomposition link refines a task into one or more intentional subelements. To perform the higher task, all subelements need to be satisfied. Accordingly, the label that is propagated to the higher task through the decomposition link is the *minimum* value among the values associated with the sub elements. In means-end relationships, the subtasks are alternative ways to achieve the higher level goal; thus means-end links work as OR relations, and the *maximum* value among alternative tasks is propagated to the goal.

5.2 Vulnerability exploits and effects propagation rules

In order to propagate the satisfaction or denial labels of malicious tasks to other elements, vulnerability is treated as a *filter*: when the filter is open, a backdoor to the system is open for attackers. To determine if the filter is open or not, both the vulnerable element and the malicious task that exploits the vulnerability need to be analyzed. If the vulnerable task is not executed, the vulnerability cannot be exploited. Similarly, if the resource that brings the vulnerability to the system is not available, then the vulnerability does not exist within the system. At the same time, a vulnerability is exploited if the attack succeeds, indicating that the malicious task has been performed. To determine the exploitation condition of a vulnerability (i.e., if it has been exploited or not), we introduce the function

$$\mathrm{exp} : MT \times TR \longrightarrow V_f$$

where MT is the evaluation value associated with the malicious task, TR is the value associated with the task or resource that has brought the vulnerability, and V_f represents the exploitation condition of the vulnerability. The evaluation value for V_f is calculated as $V_f = min\ (MT, TR)$. The satisfaction label (S) for the V_f means that the vulnerability is fully exploited, i.e., the filter is completely open. The denial label (D) indicates that the vulnerability is not exploited and the backdoor to the system is completely closed. Partial labels for V_f indicate that there exist partial evidences about the condition of the filter.

Table 2 Propagation rules for contribution links

Source label	Contribution link type				
Label name	++	+	−−	−	?
Satisfied (S)	S	PS	D	PD	?
Partially satisfied (PS)	PS	PS	PD	PD	?
Conflict (C)	C	C	C	C	?
Unknown (?)	?	?	?	?	?
Partially denied (PD)	PD	PD	PS	PS	?
Denied (D)	D	PD	PS	PS	?

The effect of the vulnerability on other intentional elements is computed on the basis of the exploitation condition of the vulnerability and the severity of its effect on the system. For instance, if a vulnerability has not been exploited, its negative effect would not be propagated to other elements. For this purpose, we employ the function

$$\mathrm{eff} : V_f \times I \longrightarrow E$$

where V_f is the exploitation condition of the vulnerability, I represents the severity of the vulnerability effect, and E represents the evidences about the satisfaction or denial of the intentional elements that is affected by the vulnerability. This function shares the same intuition of the propagation rules described in Table 2. In the case, where an attack exploits the combination of two or more vulnerabilities, human judgment is required to evaluate function eff for several combined vulnerability effects.

5.3 Countermeasure impacts propagation rules

Countermeasures can have three different security impacts: they can be used to prevent execution of a task, achievement of a goal, or the availability of a resource; patching vulnerabilities; or alleviating their effects. The propagation of the impact of a countermeasure through a *prevent* link depends on the successful employment of the countermeasure as well as on its efficacy. To evaluate the final impact of the countermeasure on the target element, we introduce the function

$$\mathrm{pre} : C \times P \longrightarrow E_{(\mathrm{new})}$$

where C is the evaluation label associated with countermeasure, P is the type of the prevent relation, and $E_{(\mathrm{new})}$ is the new evaluation value of the target intentional element affected by the *prevent*. A prevent relation shares the same nature with contribution relations, and accordingly, function *pre* uses the propagation rules defined in Table 2.

When a countermeasure patches a vulnerability, the vulnerability exploitation condition is modified. The objective of patching a vulnerability is to make the filter closer and consequently to reduce the impact of the attack that exploits the vulnerability:

$$\mathrm{patch} : C \times P \longrightarrow V_{(\mathrm{new})}$$

where C is the countermeasure evaluation label, P is the contribution type of the patch relation, and $V_{(\mathrm{new})}$ is the new value to be associated to the vulnerability. Similarly, to prevent relations, a patch relation shares the same intuition with a contribution link, and the corresponding propagation rules are similar to the ones defined in Table 2. However, we assume that countermeasures only reduce the risks and do not magnify vulnerabilities or attacks. Therefore, propagation rules apply in cases where the countermeasure is partially or fully satisfied, and the impact is not propagated if the countermeasure is partially or fully denied. Once the impact of a countermeasure is propagated through the patch link, the new exploitation condition of the patched vulnerability needs to be propagated to other instances of the vulnerability through decomposition and dependency links.

Finally, a countermeasure may alleviate the effect of an exploited vulnerability. In this case, the contribution value of the alleviate link is combined with the countermeasure evaluation value and the current effect of the vulnerability. This can be represented by the function

$$\mathrm{all} : C \times A \times E_{(\mathrm{old})} \longrightarrow E_{(\mathrm{new})}$$

where C is the countermeasure evaluation label, A is the contribution type of the alleviation relation, and $E_{(\mathrm{old})}$ and $E_{(\mathrm{new})}$ are the contribution of the vulnerability effect before and after applying the countermeasure, respectively. Table 3 defines the rules used by function *all* to compute the $E_{(\mathrm{new})}$ value.

5.4 Evaluation example

The first step of the evaluation is assignment of evaluation labels to the leaf nodes of malicious and non-malicious actors. Attacker profiles are used for the assignment of evaluation labels to the leaf nodes of malicious actors. Figure 10 shows the result of label propagation on a fragment of Fig. 6. The steps of propagation are depicted by the tag numbers assigned to the evaluation labels of each element. After the initial label assignment, labels are propagated to the upper nodes. For example, the *Malicious script* vulnerability attached to *Run the JavaScript with user inputs* task is fully exploited, because the vulnerable and malicious tasks that exploit it are fully satisfied. The exploitation condition of the vulnerability together with its effect makes *Protect users' cookies*, fully denied.

Table 3 Propagation rules for alleviate links

Countermeasure label	Alleviate contribution	Old vulnerability effect	New vulnerability effect
S	–	–	–
PS	– –	– –	–
S	– –	– –	? (No impact)
PS	– –	– –	–
S	–	– –	–
PS	–	– –	–
S	– –	–	? (No impact)
PS	– –	–	–

In Fig. 11, two alternative countermeasures, *Disable JavaScript* and *Anti Phishing tool bar*, are added to the system to analyze their impacts on the system security. Assuming that the user *Disables JavaScript* option, the evaluation process continues by propagating the impact of this countermeasure to the *Run the JavaScript with user inputs* task. As a result, the task that brings the vulnerability is fully denied, and the impact of the vulnerability is not propagated to *Protect users' cookies* goal. On the other hand, the *Build customized page* softgoal is partially denied.

6 Related work

Security requirements intend to protect the system against threats and prevent the exploitation of vulnerabilities by attackers. Security Requirements Engineering thus provides techniques for modeling and analyzing attacks, attackers and vulnerabilities, and eliciting countermeasures. In this section, we overview the current state of the art in threat analysis and security requirements engineering. We compare the proposed approach with other existing methods that analyze vulnerabilities for security requirements engineering.

6.1 Threat analysis

In security engineering, various modeling techniques have been proposed to analyze the system from the perspective of attackers [10, 22, 23, 33]. Schneier [10] proposes attack tree as a formal and methodical way for analyzing attacks. The root node of an attack tree is the goal of the attacker that is refined using AND/OR relations to understand the possible alternatives used by the attacker to achieve his goal. Attack trees can be also annotated with properties of attackers (e.g., skill, access, risk aversion, etc.) and labels representing the cost or probability of achieving a goal. Such properties allow designers to analyze the behavior of classes of attackers by focusing on a certain parts of the attack tree. Attack trees, however, are not linked to other development artifacts, such as design, architecture, and requirements specifications.

Fault Trees Analysis (FTA) [33] is one of the most commonly used techniques in reliability engineering. The main goal is to assess the likelihood of system failures based on the likelihood of external events. Fault trees visually model logical relationships among infrastructure failures, human errors, and external events which could lead to the system failure. Although FTA enables modeling faults and tracing them to events or errors, it does not provide means to express vulnerabilities of the system and link attacks to them. Moreover, FTA does not support the analysis of the impact of countermeasures on the system.

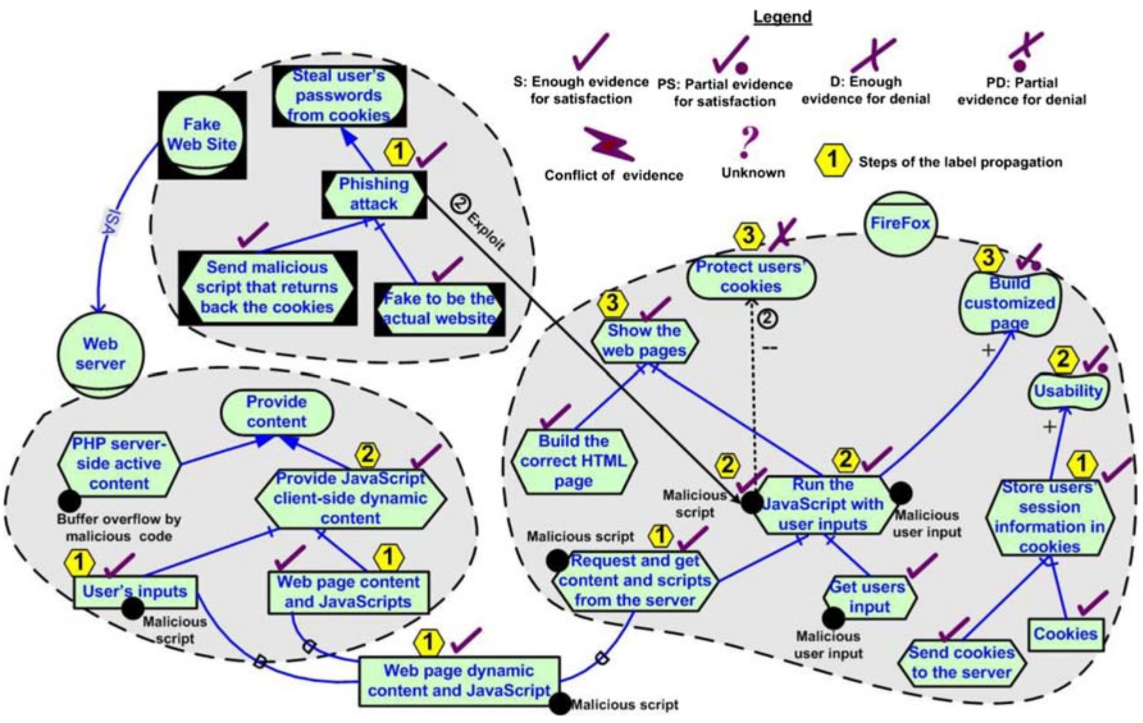
Legend
S: Enough evidence for satisfaction
PS: Partial evidence for satisfaction
D: Enough evidence for denial
PD: Partial evidence for denial
Conflict of evidence
Unknown
Steps of the label propagation
Fake Web Site
Steal user's passwords from cookies
Phishing attack
Send malicious script that returns back the cookies
Fake to be the actual website
Exploit
ISA
Web server
Provide content
PHP server-side active content
Buffer overflow by malicious code
Provide JavaScript client-side dynamic content
User's inputs
Malicious script
Web page content and JavaScripts
Web page dynamic content and JavaScript
Malicious script
FireFox
Protect users' cookies
Build customized page
Show the web pages
Usability
Build the correct HTML page
Run the JavaScript with user inputs
Malicious user input
Request and get content and scripts from the server
Get users input
Store users' session information in cookies
Send cookies to the server
Cookies

Fig. 10 Propagation of the evaluation labels through the attacker template view

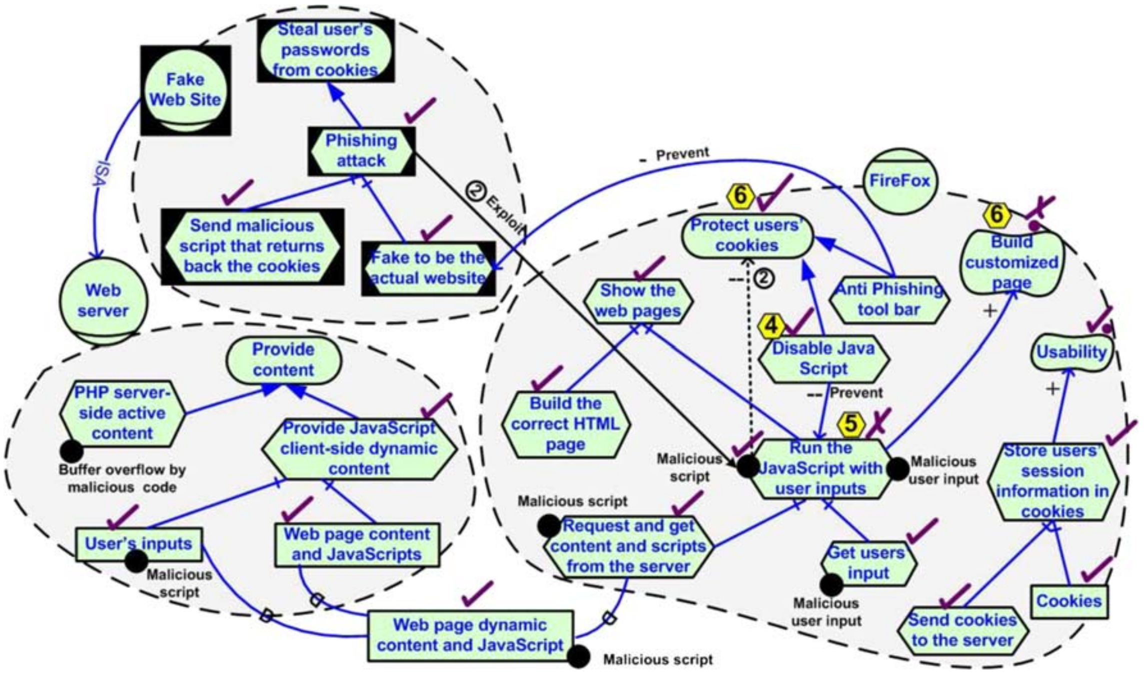
Fake Web Site
Steal user's passwords from cookies
Phishing attack
Send malicious script that returns back the cookies
Fake to be the actual website
Exploit
ISA
Prevent
Web server
Provide content
PHP server-side active content
Buffer overflow by malicious code
Provide JavaScript client-side dynamic content
User's inputs
Malicious script
Web page content and JavaScripts
Web page dynamic content and JavaScript
Malicious script
FireFox
Protect users' cookies
Anti Phishing tool bar
Disable Java Script
Prevent
Build customized page
Show the web pages
Usability
Build the correct HTML page
Run the JavaScript with user inputs
Malicious user input
Request and get content and scripts from the server
Get users input
Store users session information in cookies
Send cookies to the server
Cookies

Fig. 11 Propagation of the evaluation labels through the countermeasure view

McDermott [22] proposes to model attack nets as Petri Nets, where places represent states or modes of the security-relevant entities within the system, and transitions represent input events, commands, or data that cause one or more security-relevant entities to change state. Attack steps are represented by places, transitions are used for the explicit modeling of attacker actions, and tokens are used to indicate the progress of the attack. However, Petri Net models do not support the modeling and analysis of vulnerabilities and countermeasures. The attack model is not linked to other development artifacts, which is an obstacle to elicit security requirements and design the architecture. Although it is possible to express attackers, their goals, skills, and capabilities cannot be expressed.

Phillips et al. [23] introduced attack graphs to analyze vulnerabilities in computer networks. Attack graphs provide a method for modeling attacks and relating them to the machines in a network and to the attackers. The proposal is based on attack templates, attack profiles, and network configurations. Attack templates describe generic steps in known attacks and conditions which must be hold. The underlying idea is to match the network configuration, attacker profile, and attack templates to generate the attack graph. An attack graphs is an attack template instantiated with particular attackers/users and machines. Thereby, one can analyze an attack graph by identifying the attack paths that are most likely to succeed. Although attack graphs are able to model the steps of an attack, post and pre conditions, required configurations, and capabilities, they do not express the impact of the attacks on system functionalities.

The CORAS project [15, 16] proposes a modeling framework for model-based risk assessment in the form of a UML-profile. The profile defines UML stereotypes and rules to express assets, risks targeting the assets, vulnerabilities, accidental and deliberate threats, and the security solutions. In addition to the UML-profile, CORAS defines a methodology based on the Unified Process for risk assessment. The analysis method consists of analyzing the target context by developing asset and threat models. The potential attackers, who impose the risks, and vulnerabilities that are exploited by attackers are identified. Risks are prioritized with respect to their severity and likelihood, and the treatments for those risks that are not acceptable are identified. CORAS provides a way for expressing how a vulnerability leads to another vulnerability and how a vulnerability or combination of vulnerabilities lead to a threat. CORAS also provides facilities to relate treatments to threats and vulnerabilities. However, it does not investigate which design choices, requirements, or processes has brought the vulnerabilities to the system.

Various security and attack patterns provide knowledge repositories for analyzing and incorporating vulnerabilities, attacks, and security countermeasures into system analysis. For example, Hoglund and McGraw [34] list extensive examples and “attack patterns” which help to understand how attackers analyze software vulnerabilities and use the results of the analysis to attack systems. Whittaker et al. [35] present guidelines for software testers for uncovering security holes caused by software dependencies, data-dependent weaknesses in software, application design flaws, and implementation-related vulnerabilities. Schumacher et al. [36] provide a general overview on security patterns at the level of system and enterprise architecture. However, these proposals address security using during the design of the IT system and do not support the elicitation of security requirements.

6.2 Security requirements engineering

In recent years, the necessity of considering security from the early phases of the software development process has been recognized. To address this need, traditional requirements engineering framework has been adopted and adapted to support the modeling and analysis of security requirements. Van Lamswerde extended KAOS [37], a goal-oriented requirements engineering methodology, by introducing the notions of obstacle to capture exceptional behaviors [38] and anti-goal to model intentional obstacles set up by attackers to threaten security goals [12]. Anti-goals are defined as the negation of application-specific instances of generic security goals such as confidentiality, availability, and privacy. Anti-goals represent the goals of attackers. Anti-goals are then refined to form a threat tree on the basis of attackers’ goals and capabilities as well as software vulnerabilities. The leaf nodes are either software vulnerabilities or anti-requirements (i.e., anti-goals that are realizable by some attacker). Security requirements are defined as the countermeasures to software vulnerabilities or anti-requirements. The framework does not consider assets as a main concept for eliciting and elaborating security requirements. In addition, vulnerabilities are identified as part of the anti model, while vulnerabilities exist independently from the threats.

A number of approaches based on i*/Tropos [28, 39] have been adopted to address different security aspects. Liu et al. [8] use the i* framework to model the relationships among strategic actors explicitly for eliciting, identifying, and analyzing security requirements. In this approach, all actors are assumed potential attackers, which inherit capabilities, intentions, and social relationships of the corresponding legitimate actor. The framework attempts to identify the vulnerable points in the dependency network when an actor behaves maliciously and to understand the measures necessary to protect the system. Attacker identification, however, is limited to analyzing what roles in the system can impose threats on the dependencies and ignores external attackers. Moreover, the approach does not explicitly describe how countermeasures need to be incorporated into the model and what are their impacts on attacks and other goals.

A different perspective has been adopted by Massacci et al. [25] who define Secure Tropos for modeling and analyzing authorization, trust, and privacy concerns. Secure Tropos extends Tropos with concepts specific to security, namely ownership, permission, delegation, and trust. These constructs have be proved to be expressive enough to capture privacy-related legal requirements [27] and have been used to define and validate security and dependability patterns [40]. In [41], the authors shows how Secure Tropos can be applied to design and analyze access control policies. Secure Tropos, however, addresses security issues within the organization setting rather than dealing with malicious actors and system vulnerabilities.

Mouratidis and Giorgini [42] introduce extensions to the Tropos methodology for incorporating security concerns into the agent-oriented development process and modeling security concerns throughout the whole development process. In this approach, security features of the system-to-be, the protection objectives of the system, the security mechanisms, the threats to the system's security features, and security requirements as constraints are modeled. However, similarly to [25], attackers and their malicious goals and tasks, vulnerabilities, and vulnerability exploitations are not modeled and analyzed.

Some proposals focus on integrating risk analysis into the requirements engineering mainstream. Asnar et al. [43] propose Goal-Risk (GR) Tropos, which extends Tropos with three basic layers: strategy, event, and treatment. The strategic layer analyzes strategic interests of the stakeholders; the event layer analyzes uncertain events along their impacts to the strategy layer; and treatment layer analyzes treatments to be adopted in order to mitigate risks. In [13], GR-Tropos has been extended to assess risk in organizational settings. However, the framework mainly concerns the development of safety critical system and does not consider the intentionality of attackers.

Mayer et al. [44] analyze the impacts of risks on business assets and elicit security requirements for risk mitigation. In this work, risks are related to threats and vulnerabilities in the architecture. In a more recent work, Mayer et al. [45] investigates the necessary concepts in modeling language for the purpose of information system security risk management. They propose a meta-model based of five main concepts: *risk*, *cause of the risk*, *impact*, *threat*, and *vulnerability* in this context. In [46], Mayer et al. enrich the proposed meta-model by adding measurement metrics to the meta-model.

Similar to our approach, threats are related to vulnerabilities, and their exploitation has some impacts. However, in [44, 45], threats are not assigned to an actor and vulnerabilities are not attached to actions of the actors that bring the vulnerability to the system; also the impact of countermeasures on the vulnerabilities is not considered as part of the meta-model.

Matulevicius et al. [14] improves the Secure Tropos [42][3] modeling language for risk management purposes, where risk is defined as the combination of a threat with vulnerabilities leading to negative impacts on assets. Vulnerabilities are treated as beliefs inside the boundary of attackers which may contribute positively to the successful of an attack. However, similar to the CORAS framework [15], the resulting model does not specify how the vulnerability is brought to the system, by what actions, and by what actors. In addition, the enhanced Secure Tropos models do not capture the impact of countermeasures on the vulnerabilities and attacks.

Haley et al. [47] propose a security requirements framework based on constructing the system context, representing security requirements, and developing satisfaction arguments for those requirements. The framework extends problem frames and intends to determine adequate security requirements for the system by considering threats as crosscutting concerns. Functional requirements describe how assets (i.e., objects to be protected) are used within the system, and threats describe how attackers can compromise the security of assets. Security requirements are thus defined as constraints on functional requirements. Once security requirements are elicited, satisfaction arguments are used to verify that security requirements are satisfied by the system as described by the context. This proposal, however, mainly focuses on system requirements and does not provide methodological support for the analysis of the organizational context where the system will operate.

In the UML community, Sindre and Opdahl [11] propose analyzing security requirements by defining misuse cases, inverted UML use cases, which describe functions that the system should not allow. They are depicted as black ovals to distinguish them from traditional use cases. Misuse cases can be linked to use cases to indicate that the use case is exploited by the misuse case, and use cases to misuse cases to indicate that the use case is a countermeasure against the misuse case. This new construct makes it possible to represent actions that the system should prevent together with those actions which it should support. A similar proposal is defined by McDermott and Fox [48], who introduce abuse cases to specify the interactions that their results are harm to system. Differently form misuse cases, abuse cases are distinguished from use cases by representing them in separated models. This does not allow one to analyze the impact of an abuse case on use cases. The security requirements elicitation process underling abuse and misuse cases does not consider why and how security goals are defined without analyzing what may threaten the assets.

Rostad [49] suggests extending the misuse case notation for including vulnerabilities into requirements models. Vulnerabilities are defined as a weakness that may be exploited by misuse cases. Vulnerabilities are expressed as a type of use case, with an *exploit* relationship from the

[3] In security requirements literature, two different frameworks developed by different researchers are called Secure Tropos [25, 42].

misuse case to the vulnerability and an *include* relation with the use case that introduces the vulnerability. However, the semantics of the countermeasure impact is not well defined, and the model cannot be used to evaluate the impact of countermeasures on the overall security of the system.

Jürjens proposes UMLsec [7], a UML-profile designed to express security-relevant information within UML diagrams. UMLsec objectives are to encapsulate knowledge and make it available to developers in the form of a widely used design notation, and to provide formal techniques for the verification of security requirements. The profile is described in terms of UML stereotypes, tags, and constraints that can be used in various UML diagrams such as activity diagrams, statecharts, and sequence diagrams. The stereotypes and tags encapsulate the knowledge of recurring security requirements of distributed object-oriented systems, such as secrecy, fair exchange, and secure communication link. Assigning a stereotype and tag to the model and defining potential threats make it possible to analyze the behavior of subsystems in order to verify the security impact of threats on the system. The defined security requirements are high level and general. This increases the reusability of the extensions in various contexts. On the other hand, the default threats may not hold in every context. UMLsec has been used for model-based security testing [50] as well as to analyze whether workflows and the design of the security permissions for the system fit together [51]. However, UMLsec has its strength during the system design phase, but it lacks the support for the elicitation of security requirements during the early phases of the development process.

6.3 Discussion and comparison

Table 4 compares capabilities of existing vulnerabilities conceptual structures based on the conceptual foundation discussed in Sect. 2. Few security modeling notations provide explicit constructs for modeling vulnerabilities and analyzing their impacts on security requirements. As mentioned previously, CORAS framework models and analyzes vulnerabilities by linking them to the threats and risks. In [14], vulnerabilities are modeled as beliefs in the attackers knowledge boundary which may contribute positively to the attacks. In [44, 45], i* is extended to represent vulnerabilities and their relation with threats and other elements of the i* goal model. Rostad [49] suggests extending the misuse case notation for including vulnerabilities into requirements models. Vulnerabilities are defined as a weakness that may be exploited by misuse cases. Vulnerabilities are expressed as a type of use case, with an *exploit* relationship from the misuse case to the vulnerability and an *include* relation with the use case that introduces the vulnerability.

The missing point in these approaches is providing modeling constructs to understand why vulnerabilities are within the system and how they are spread out among the actors. To our knowledge, existing modeling notations do not provide means to assign vulnerabilities to the actions that actors perform or assets they use. The harmful effects of vulnerabilities on stakeholders and system requirements are not expressed by existing proposals. Among them, CORAS [16] does not investigate which design choices, requirements, or processes has brought the vulnerabilities to the system, and semantics of relationships between vulnerabilities, and between vulnerabilities and threats are not defined. The semantics of the countermeasure impact in [16, 49] is not well defined, and the model cannot be used to evaluate the impact of countermeasures on the overall security of the system. Similar to CORAS, the resulting models in [14, 44, 45] do not specify how the vulnerability is brought to the system, by what actions, and by what actors. The proposals in [14, 44, 45] do not capture the impact of countermeasures on the vulnerabilities and attacks. In [44, 45], threats are not related to the attacker that poses them, and the semantics of the relation between threats and vulnerabilities are not defined. Finally, modeling and analyzing the steps and the order of actions to accomplish an attack affect the countermeasure selection and development, the existing conceptual modeling frameworks for security requirements engineering do not incorporate the concept of time into their meta-model.

7 Conclusions and future work

This paper proposes a requirements engineering framework to support the elicitation of security requirements based on the vulnerabilities that requirements and design decisions bring to the system. The framework comprises a modeling framework that extends i* with the concept of vulnerability and relations that allow modeling and understanding effects of vulnerabilities on security requirements. Security requirements are expressed in the form of countermeasures to be adopted to prevent attacks, patch vulnerabilities, or alleviate their effect. Together with a modeling notation, the framework provides an evaluation method for assessing vulnerabilities risks and countermeasures efficacy.

This work is still in progress to better support system designers in modeling and analysis of security requirements. Goal models and the i* modeling notation do not capture temporal aspects of the systems. Therefore, the resulting models do not provide a temporal sequence to guide reading and understanding the model for the analysts that view the large and complicated models. A major limitation of the proposed approach concerns scalability issues. Goal models contain multiple actors and

Table 4 Comparison of vulnerability modeling and analysis approaches

Method	CORAS Framework [5]	Secure Tropos [27]	Risk-Based Security Framework [29, 30]	Extensions to misuse case diagram [37]	Current proposal
Conceptual Foundation	CORAS UML-profile based models	Secure Tropos	i* framework	Misuse case models	i* framework
Vulnerability graphical representation	🔓	Belief	◇	Vulnerability	●
Relation of vulnerabilities to vulnerable elements	N	N	N	P	Y
Relation of vulnerabilities to other vulnerabilities	P	N	N	N	N
Propagation of vulnerabilities to the other system elements	N	N	N	N	Y
Effects of vulnerabilities	Y	Y	P	N	Y
Severity of vulnerabilities	N	N	N	N	Y
Relation of vulnerabilities and attacks (exploitation)	P	P	Y	Y	Y
Countermeasures' impacts on vulnerabilities	P	N	N	Y	Y
Steps of vulnerability exploitation (time)	N	N	N	N	N

N indicates that the concept or relation is not considered, *Y* indicates the relation is considered explicitly in the notation, *P* the relation is implicitly considered or its semantics is not well defined

dependency chains, and each actor includes several intentional elements and complicated relationships. The resulting models, especially extended with security concepts, can be complicated and hard to understand. This requires development of modeling and analysis tools that provide resolution management and handle the model complexity by providing views of the security requirements model. To manage the complexity of the models, one can filter vulnerabilities that are not exploited as well as their effects. Analysts would benefit from views that focus on a specific attack, vulnerability, or countermeasure of importance, which cuts some other elements out of the model.

The proposed framework assumes that analysts have knowledge about vulnerabilities, potential attacks, and proper countermeasure or can obtain such information. In particular, the analysis of vulnerabilities, such as propagating them through the goal model or identifying their impacts requires experience with vulnerable software products and services. However, existing vulnerability databases do not provide the required knowledge for linking vulnerabilities to actions and assets of the system actors and to potential attacks and countermeasures. Therefore, software developers without security expertise may need additional support for applying the proposed framework.

To address these issues, we are building catalogs of attacker templates that defines the behavior of attackers and catalogs of countermeasures that describe how security flaws are addressed in current security practices. The attacker catalogs assume limited skills and capabilities for the attackers and analyze the actions that they can perform for compromising the system in detail. The attacker templates are then instantiated using attacker profiles to study the behavior of particular classes of attackers. We are also developing security metrics based on the risk attitude of system designers and stakeholders as well as on specific application domains to assist designers during the decision making process.

Finally, we are currently performing empirical studies for evaluating the expressiveness of the proposed modeling notation and the accuracy of the analysis method. Human subjects are being asked to use the proposed framework for modeling and analyzing a number of case studies. The modelers are interviewed, and models are critically analyzed to draw conclusions about the practical usefulness and expressiveness of the approach.

Acknowledgments Financial support from Natural Science and Engineering Research Council of Canada and Bell University Labs is gratefully acknowledged.

References

1. Anderson R (2001) Security engineering: a guide to building dependable distributed systems. Wiley, London

2. IBM Global Technology Services (2008) IBM internet security systems X-force 2007 trend statistics
3. National Vulnerability Database. http://www.nvd.nist.gov/
4. SANS. http://www.sans.org/
5. Common Weakness Enumeration. http://www.cwe.mitre.org/
6. Common Vulnerability Scoring System. http://www.first.org/cvss/
7. Jürjens J (2004) Secure systems development with UML. Springer, Berlin
8. Liu L, Yu E, Mylopoulos J (2003) Security and privacy requirements analysis within a social setting. In: Proceedings of the 11th IEEE international conference on requirements engineering. IEEE Computer Society, pp 151–161
9. Giorgini P, Massacci F, Mylopoulos J, Zannone N (2006) Requirements engineering for trust management: model, methodology, and reasoning. Int J Inf Secur 5(4):257–274
10. Schneier B (1999) Attack trees. Dr. Dobb's J 24(12):21–29
11. Sindre G, Opdahl AL (2005) Eliciting security requirements with misuse cases. Requir Eng 10(1):34–44
12. van Lamsweerde A (2004) Elaborating security requirements by construction of intentional anti-models. In: Proceedings of the 26th international conference on software engineering. IEEE Computer Society, pp 148–157
13. Asnar Y, Moretti R, Sebastianis M, Zannone N (2008) Risk as dependability metrics for the evaluation of business solutions: a model-driven approach. In: Proceedings of the 3rd international conference on availability, reliability and security. IEEE Computer Society, pp 1240–1248
14. Matulevicius R, Mayer N, Mouratidis H, Dubois E, Heymans P, Genon N (2008) Adapting secure tropos for security risk management in the early phases of information systems development. In: Proceedings of the 20th international conference on advanced information systems engineering, LNCS 5074. Springer, pp 541–555
15. Braber F, Hogganvik I, Lund MS, Stolen K, Vraalsen F (2007) Model-based security analysis in seven steps—a guided tour to the CORAS method. BT Technol J 25(1):101–117
16. den Braber F, Dimitrakos T, Gran BA, Lund MS, Stolen K, Aagedal JO (2003) The CORAS methodology: model-based risk assessment using UML and UP. In: UML and the unified process. IGI Publishing, Hershey, PA, pp 332–357
17. Elahi G, Yu E (2007) A goal oriented approach for modeling and analyzing security trade-offs. In: Proceedings of 26th international conference on conceptual modeling, LNCS 4801. Springer, pp 375–390
18. ISO/IEC (2004) Management of information and communication technology security—part 1: concepts and models for information and communication technology security management. ISO/IEC 13335
19. Kissel ER (2005) Glossary of key information security terms. NIST IR 7298
20. Schneider FB (ed) (1998) Trust in cyberspace. National Academy Press, Washington
21. Schneier B (2003) Beyond fear. Springer, Berlin
22. McDermott JP (2000) Attack net penetration testing. In: Proceedings of the 2000 workshop on new security paradigms. ACM, pp 15–21
23. Phillips C, Swiler LP (1998) A graph-based system for network-vulnerability analysis. In: Proceedings of the 1998 workshop on new security paradigms. ACM, pp 71–79
24. Avizienis A, Laprie J-C, Randell B, Landwehr CE (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33
25. Massacci F, Mylopoulos J, Zannone N (2008) An ontology for secure socio-technical systems. In: Handbook of ontologies for business interaction, Chap. XI. The IDEA Group
26. Schneier B (2007) The psychology of security. Commun ACM 50(5):128
27. Massacci F, Prest M, Zannone N (2005) Using a security requirements engineering methodology in practice: the compliance with the Italian data protection legislation. Comp Stand Interf 27(5):445–455
28. Yu ESK (1995) Modeling strategic relationships for process reengineering. PhD thesis, University of Toronto
29. Sindre G, Opdahl AL (2007) Capturing dependability threats in conceptual modelling. In: Conceptual modelling in information systems engineering. Springer, pp 247–260
30. Yu ESK (1997) Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE international conference on requirements engineering. IEEE Computer Society, pp 226–235
31. Chung L, Nixon BA, Yu E, Mylopoulos J (eds) (2000) Non-functional requirements in software engineering. Kluwer, Dordrecht
32. Horkoff J (2006) Using i* models for evaluation. Master's thesis, University of Toronto
33. Vesely WE, Goldberg FF, Roberts N, Haasl DF (1981) Fault tree handbook. Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission
34. Hoglund G, McGraw G (2004) Exploiting software: how to break code. Addison-Wesley Professional, Reading
35. Whittaker JA, Thompson H, Thompson HH, Thompson H (2003) How to break software security: effective techniques for security testing. Pearson
36. Schumacher M, Fernandez-Buglioni E, Hybertson D, Buschmann F, Sommerlad P (2006) Security patterns: integrating security and systems engineering. Wiley, London
37. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Program 20:3–50
38. van Lamsweerde A, Letier E (2000) Handling obstacles in goal-oriented requirements engineering. IEEE Trans Softw Eng 26(10):978–1005
39. Bresciani P, Giorgini P, Giunchiglia F, Mylopoulos J, Perini A (2004) TROPOS: an agent-oriented software development methodology. J Auton Agents Multi-Agent Syst 8(3):203–236
40. Compagna L, Khoury PE, Krausová A, Massacci F, Zannone N (2009) How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. Artif Intell Law 17(1):1–30
41. Massacci F, Zannone N (2008) A model-driven approach for the specification and analysis of access control policies. In: Proceedings of the OTM 2008 confederated international conferences, LNCS 5332. Springer, pp 1087–1103
42. Mouratidis H, Giorgini P (2007) Secure tropos: a security-oriented extension of the tropos methodology. Int J Softw Eng Knowl Eng 17(2):285–309
43. Asnar Y, Giorgini P (2006) Modelling risk and identifying countermeasure in organizations. In: Proceedings of the 1st international workshop on critical information infrastructures security, LNCS 4347. Springer, pp 55–66
44. Mayer N, Rifaut A, Dubois E (2005) Towards a risk-based security requirements engineering framework. In: Proceedings of the 11th workshop on requirements engineering for software quality
45. Mayer N, Heymans P, Matulevicius R (2007) Design of a modelling language for information system security risk management. In: Proceedings of the 1st international conference on research challenges in information science, pp 121–132
46. Mayer N, Dubois E, Matulevicius R, Heymans P (2008) Towards a measurement framework for security risk management. In: Proceedings of modeling security workshop, 2008

47. Haley C, Laney R, Moffett J, Nuseibeh B (2008) Security requirements engineering: a framework for representation and analysis. IEEE Trans Softw Eng 34(1):133–153
48. McDermott J, Fox C (1999) Using abuse case models for security requirements analysis. In: Proceedings of the 15th annual computer security applications conference. IEEE Computer Society, pp 55–66
49. Rostad L (2006) An extended misuse case notation: including vulnerabilities and the insider threat. In: Proceedings of the 12th working conference on requirements engineering: foundation for software quality
50. Jürjens J (2008) Model-based security testing using UMLsec: a case study. Electron Notes Theoretical Comput Sci 220(1):93–104
51. Jürjens J, Schreck J, Yu Y (2008) Automated analysis of permission-based security using UMLsec. In: Proceedings of 11th international conference on fundamental approaches to software engineering, LNCS 4961. Springer, pp 292–295