Security Requirements for the Rest of Us: A Survey

Inger Anne Tøndel, Martin Gilje Jaatun, and Per Håkon Meland, Sintef ICT

Most software developers aren’t primarily interested in (or knowledgeable about) security. For decades, the focus has been on implementing as much functionality as possible before the deadline, and patching the inevitable bugs when it’s time for the next release or hot fix. However, the software engineering community is slowly beginning to realize that information security is also important for software whose primary function isn’t related to security. Security features or mechanisms typically aren’t prominent in such software’s user interface, but a persistent search of menus and configuration options normally uncovers some security-relevant items.

A 2006 Computer Security Institute/US Federal Bureau of Investigation survey reports that 131 respondents estimated their losses due to computer security incidents in 2006 to exceed US$50 million. If we eliminate causes such as laptop theft and run-of-the-mill virus infections, we’re left with losses of about $30 million. So, the average organization experiences a setback of about $230,000 a year. Some of this is probably due to configuration error, but we believe that software security flaws cause much of this cost.

Because security is important for ordinary software development projects, we need mechanisms for security requirements elicitation that will be palatable to regular software developers and suitable for use in all software development. These mechanisms must be easy to both understand and use. Although formal methods undoubtedly have their merits, their use is precluded in this context.

Numerous publications underline security requirements’ importance, but few provide advice that’s concrete and specific enough for immediate deployment. Our impression is that people are saying a lot about security requirements but aren’t doing much to elicit such requirements in software development projects.

We surveyed the literature to identify and describe concrete techniques for eliciting security requirements. For our purposes, a technique is a series of well-defined steps that collectively lead to the elicitation of security requirements. We sketch the outline of a new approach, and conclude with our most important conviction: All software development projects need a well-balanced amount of security awareness right from the beginning.

Security requirements in the literature

Donald Firesmith claims that most requirements engineers are poorly trained to elicit, analyze, and specify security requirements. Consequently, they often confuse security requirements with architectural security mechanisms that are traditionally used to fulfill requirements, and end up making architecture and design decisions. Charles Haley and
his colleagues recognize the same problem. They show that several standards (such as the Common Criteria and the US National Institute of Standards and Technology computer security handbook) suggest describing security requirements in terms of security mechanisms. However, as they point out, “Defining requirements in terms of function leaves out key information: what objects need protecting and, more importantly, why the objects need protecting.”

Haley and his colleagues define security requirements as “constraints on the functions [that] ... operationalize one or more security goals.” They take issue with those who specify security requirements as high-level security goals, arguing that this makes it difficult to make the requirements specific enough to guide designers and to verify that the requirements are met. For the same reason, they recommend security requirements that “express what is to happen in a given situation, as opposed to what is not ever to happen in any situation.”

The Comprehensive Lightweight Application Security Process (CLASP, www.fortifysoftware.com/security-resources/clasp.jsp) states that all requirements should be SMART+ requirements: specific, measurable, appropriate, reasonable, and traceable. CLASP gives no examples, however, as to what a typical security requirement should look like. Firesmith gives such examples. He defines a security requirement as “a detailed requirement that implements an overriding security policy.” He suggests dividing security requirements into categories, such as identification, integrity, and privacy requirements. For example, the requirement “The application shall identify all of its client applications before allowing them to use its capabilities” is an identification requirement, whereas “The application shall not allow unauthorized individuals or programs access to any communications” is a privacy requirement.

Haley and his colleagues also give examples of security requirements, such as “The system shall provide Personnel Information only to members of Human Resources Dept.” By expressing security requirements in relation to specific functional requirements, they claim that they can achieve enough specificity to guide designers and let them verify that the requirements are actually fulfilled.

These examples notwithstanding, we haven’t found a universally accepted definition of “security requirement” in the literature. We’ll return to this point later.

**How to proceed**

Table 1 summarizes some major approaches to security requirements engineering, in relation to tasks recommended as part of the requirements engineering process. It compares different approaches to security requirements engineering based on how they handle the requirements phase tasks.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Definitions</th>
<th>Objectives</th>
<th>Misuse/Threats</th>
<th>Assets</th>
<th>Coding Standards</th>
<th>Categorize &amp; Prioritize</th>
<th>Inspect &amp; Validate</th>
<th>Process Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQuaRE</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charles Haley and colleagues</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gustav Boström and colleagues</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASP</td>
<td>√</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Axelle Apvrille and Makan Pourzandi</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eduardo Fernandez</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kenneth van Wyk and Gary McGraw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gunnar Peterson</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Not part of the Comprehensive Lightweight Application Security Process green-field roadmap

Part of threat modeling specified for the design phase
Fulfilling a security requirement might lead to new assets, resulting in new security requirements. We categorized the tasks on the basis of our literature survey, with a level of detail that includes all recommended tasks but avoids complexity. For a task to be part of an approach, the authors in question must consider it important enough to explicitly include it in the approach’s description. We determined eight task categories:

- definitions of central concepts (such as attack, asset, and authentication) as part of the requirements engineering process;
- objectives, or high-level business-centric goals;
- misuse or threats;
- assets and their values;
- coding standards—that is, what language to use or functions to avoid;
- categorization and prioritization of the security requirements;
- inspection and validation of requirements for completeness, lack of conflicts, or other matters; and
- process planning of security activities.

The Software Engineering Institute’s SQUARE (Secure Quality Requirements Engineering)\(^\text{10}\) has nine main steps:

1. Agree on definitions.
2. Identify security goals.
3. Develop artifacts.
5. Select an elicitation technique.
6. Elicit security requirements.
7. Categorize requirements.
8. Prioritize requirements.
9. Inspect requirements.

SQUARE is based on interaction between requirements engineers and an IT project’s stakeholders, where facilitation by a requirements engineering team is of major importance. Although SQUARE is intended for the early phases of software development, step 3 in particular requires some previous design activity. This is because the artifacts that are to be developed or identified include system architecture diagrams.

As the number of steps indicates, SQUARE covers most of the tasks in Table 1. Stating what SQUARE includes and doesn’t include isn’t straightforward, because developers can choose several techniques for the different steps. For example, the main SQUARE methodology doesn’t include asset identification, but one case study uses survivable system analysis,\(^\text{10}\) which includes identification of essential assets and services.

Haley and his colleagues’ framework gives four main steps:\(^\text{8}\)

1. Identify functional requirements.
2. Identify security goals—including assets, threats, management principles, and business goals.
3. Identify security requirements.
4. Verify the system.

The authors suggest using Jon Hall and his colleagues’ problem diagrams,\(^\text{17}\) a notation most developers are unfamiliar with. Although they also suggest that verification include formal argumentation, the main methodology would likely work with more informal techniques. Iteration between requirement and design activities is an important part of the framework. Fulfilling a security requirement might lead to new assets, resulting in new security requirements.

Gustav Boström and his colleagues consider security requirements engineering in a different context—agile development with a focus on extreme programming (XP) practices.\(^\text{11}\) They suggest seven steps for identifying and handling security requirements:

1. Identify critical assets.
2. Formulate abuser stories (that is, brief, informal descriptions of how an attacker might abuse the system).
3. Assess abuser story risk.
4. Negotiate abuser and user stories.
5. Define security-related user stories.
7. Cross-check abuser stories and countermeasures.

This process extends XP user stories to include security requirements. The main ideas should also be relevant for other types of development processes.

CLASP is a major initiative for securing software development life cycles. It specifies a set of process pieces that you can integrate into any software development process. Maybe because of CLASP’s general nature, the steps outlined aren’t as concrete as, for example, Boström and his colleagues’\(^\text{11}\) The requirements phase of the CLASP green-field roadmap has two steps (that is, steps recommended for new software development):

1. Document security-relevant requirements (for example, identify business requirements and functional security requirements and dependencies) for determining risk mitigations and resolving deficiencies and conflicts.
2. Identify resources and trust boundaries, such as network-level design and data resources.

Several authors suggest use cases as a starting point for identifying security requirements. Although the green-field roadmap doesn’t include the task “detail misuse cases,” CLASP describes this task as a possible requirements activity. Eduardo Fernandez points out that use cases are helpful for determining the rights each actor needs and for considering possible attacks. Gunnar Peterson suggests use and misuse cases as a basis for security requirements, but notes that you might need additional nonfunctional requirements. Kenneth van Wyk and Gary McGraw suggest using abuse cases. Abuse cases are also one of McGraw’s “touchpoints” for the requirements and use cases phase, together with security requirements. However, McGraw gives little detail on this security requirements touchpoint.

Steve Lipner and Michael Howard describe the Microsoft Trustworthy Computing Security Development Lifecycle, focusing on planning security activities that are to take place later in the software development life cycle. They also state that you should identify key security objectives together with security feature requirements that are based on customer demand and compliance with standards. You’ll identify other security feature requirements as part of threat modeling, which takes place during design. This corresponds with J.D. Meier’s (also from Microsoft) description of Web application security engineering. Meier describes identification of security objectives as an important activity occurring before threat modeling. Threat modeling, although part of the design phase, also has some steps that you might consider for the requirements phase:

1. Identify use scenarios.
2. Identify assets.
3. Identify threats.
4. Identify dependencies.

Axelle Apvrille and Makan Pourzandi suggest four steps for the security requirements and analysis phase:

1. Identify the security environment and objectives.
2. Determine the threat model.
3. Choose a security policy, which includes prioritizing according to the information’s sensitivity.
4. Evaluate risk.

They intend that the developers themselves perform these steps, but they don’t explain the steps in much detail.

Because we aim for a lightweight approach, this survey deliberately omits formal methods. Unfortunately, this also means that Howard Chivers’s work on automated risk analysis falls outside this article’s scope, because it relies on formal modeling. The same goes for Axel van Lamsweerde’s work on intentional antimodels.

**Security requirements artifacts**

Eliciting security requirements typically produces some artifacts.

**Misuse cases.** Guttorm Sindre and Andreas Opdahl suggest misuse cases, which have been used in several research and industrial projects. They extend the regular UML use case diagrams with misuse cases that specify behavior not wanted in the system. They define two relationships between misuse and use cases:

- Use cases can mitigate misuse cases; that is, a use case can be a countermeasure against a misuse case, thereby reducing the chances that the misuse case succeeds.
- Misuse cases can threaten use cases; that is, a misuse case can jeopardize or hinder the use case.

John McDermott and Chris Fox suggest abuse cases for expressing threats using the standard UML use case notation. Their approach keeps abuse cases in separate models. Firesmith discusses security use cases, which are similar to ordinary use cases but represent specific security functionality (countermeasures). Sindre and Opdahl and Lillian Røstad further discuss the differences between misuse cases, abuse cases, and security use cases. Røstad also suggests extending the notation for misuse cases to cover vulnerable functions and distinguish insiders from outside attackers.

**Abuser stories.** Johan Peeters introduces abuser stories as an agile counterpart to abuse cases. Abuser stories aren’t yet widely used. Boström and his colleagues demonstrate how to use them to extend the XP planning game. Vidar Kangsli reports how developers successfully used misuse stories when addressing security in a software development organization.

**Attack trees and threat trees.** Attack trees represent attacks against a system in a tree structure. The root node represents the attack’s goal, and leaf nodes
represent different ways of achieving that goal. The diagrams are useful in both the requirements and design phases. You can represent trees graphically or write them in outline form. You can also add information—for example, the attack’s cost.

Microsoft uses threat trees in its threat-modelling activity. Threat trees are similar to attack trees but add mitigated conditions to the identified threats.

Softgoal interdependency graphs. The nonfunctional requirements framework and its associated softgoal interdependency graphs provide a systematic approach to identifying system goals and decomposing them into lower-level subgoals and ultimately into potential solutions (operationalizations). Unlike goals, which are satisfied absolutely when their subgoals are identified, softgoals are “goals that do not have a clear-cut criterion for their satisfaction.” That is, softgoals are satisfied when “sufficient positive and little negative evidence for this claim” exists. Industry hasn’t yet broadly adopted the NFR framework and SIGs, but researchers have demonstrated their usefulness in several complex software systems.

Jane Cleland-Huang and her colleagues extend SIGs to represent security requirements and threats to these requirements. In their approach, you can add information for estimating potential loss, and use this to find the most efficient safeguards (calculate annual loss savings).

Comparisons

Our survey reveals that no common agreement exists on what a security requirement is. More specifically, the various approaches don’t agree on the extent to which the requirements should state concrete security measures. SQUARE requires some design work as background material for security requirements elicitation. Microsoft considers some of the other approaches’ requirement activities as part of the design phase. CLASP considers determining risk mitigations to be a requirements activity, while others consider this part of design. One reason for these differences might be different focuses on iteration. Agile development, Boström and his colleagues’ focus, is iterative, unlike what seems to be the case with other approaches, such as SQUARE.

The approaches also provide different levels of detail as to how to perform the tasks. For example, CLASP’s and Aprville and Pourzandi’s suggestions are more general, and thereby less concrete, than those of Boström and colleagues, who focus on XP development. The approaches also require different levels of expert knowledge. SQUARE relies on a requirements team facilitating the process. Haley and his colleagues suggest artifacts that are probably too complex for regular developers. Other approaches, such as those of Boström and colleagues, Microsoft, and CLASP, are more lightweight.

The assumption underlying table 1 is that the tasks that are often recommended are the most important. We’re aware of methodical limitations. The different approaches might weight the activities differently, some approaches are more accepted than others, and the comparison criteria might be inadequate. Table 1 shows that the approaches commonly recommend misuse or threat identification. Many of the typical artifacts used for security requirements engineering also support this task. Many of the approaches also recommend identifying objectives and assets. So, our impression is that these three tasks are highly important to security requirements engineering, although the approaches’ recommendations as to how to perform them differ.

Proposed approach

We aim to identify a set of security requirement techniques convenient enough for average software developers to adopt. The resulting approach probably won’t be able to identify all security requirements; we focus on the most critical ones. In an iterated development process, developers will be able to revise and add further requirements at later stages.

Our main focus is on identifying threats, assets, and security objectives, which our survey determined to be the most commonly recommended security requirement tasks. Additionally, the lack of a common definition of a security requirement indicates that developers need concrete examples for describing and documenting such requirements. We now sketch a lightweight method for security requirements engineering. Our recommendations focus on easy tasks based on our experience and on recommendations in the surveyed literature.

Security objectives

By security objectives, we mean the high-level requirements or goals that are most important to customers, and the requirements that must be met to comply with relevant legislation, policies, and standards. Developers must know these requirements to decide where to put the main effort. According to Haley and his colleagues, achieving a security objective typically involves one or more detailed security requirements. We suggest dividing this activity into two main steps:

1. Identify the customer’s need for security.
2. Identify relevant legislation, policies, standards, and best practices that apply to the system or module.

Developers need step-by-step guidelines on how to prepare for customer meetings that focus on security issues and on where to look for requirements from legislation, policies, and standards.\(^8\)

**Asset identification**

To be able to elicit and prioritize security requirements, developers need to know which properties of what assets are most important to protect. We suggest looking at the assets’ value from the customer’s, system owner’s, and attacker’s viewpoints—because, as Haley and his colleagues point out,\(^8\) different actors’ views of an asset aren’t directly related. To identify assets, you can use the system’s functional requirements, possibly combined with brainstorming. (Be aware that using functional requirements as a starting point risks not covering assets such as reputation. For this lightweight approach, however, looking at the different actors’ values for such assets can indirectly cover them.) For each asset, you should make a judgment regarding the different stakeholders’ priorities for the asset’s confidentiality, integrity, and availability. You could formalize this process by calculating the risk to an asset in the conventional manner (risk = impact × probability). But because estimating impact and probability is frequently difficult, directly choosing one of a predefined set of qualitative categories (for example, high, medium, or low) might be just as useful.

**Threat analysis**

We recommend focusing on the most important assets and using predefined threat categories such as Stride (spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege).\(^12\) You can further elaborate the most important threats identified using techniques such as attack trees. This approach will reduce the number of threats analyzed and thereby the effort needed—but always with the risk of neglecting something. We recommend attack trees, because they aren’t closely coupled to any specific development process but are common and well known.

However, if developers have already modeled the system using UML use cases, misuse cases might be preferable. Likewise, agile development projects might find abuser stories best suited. The attack trees (or any other artifact) created at this stage shouldn’t make assumptions regarding design decisions that haven’t been made. However, you can increase their level of detail later during development.

**Documentation of security requirements**

The software development community needs good descriptions of what a security requirement is. We recommend describing requirements that focus on what should be achieved—not how. This corresponds to Firesmith’s recommendations and examples.\(^7\) However, security requirements differ from high-level security goals, as Haley and his colleagues point out, because they detail “who can do what, when.”\(^8\) In our techniques, security objectives and asset identification will typically result in policy-level requirements such as, “The privacy of the service’s users must be protected.” This policy-level requirement might then result in a number of security requirements such as, “The personal profile of users must be protected in transit.” (We could probably further detail this requirement. For example, we could state when this would happen, placing it in the context of the system’s functional requirements.) We’d select the mechanisms used to achieve this protection (such as an encryption algorithm) during design.

Our goal is to create a method that lets developers, by following step-by-step guidelines, automatically elicit good-quality security requirements. Developers should then document the requirements to ensure visibility, show which security requirements have high priority, and arrange for traceability and follow-up of requirements. We suggest describing all security requirements in one place, preferably as part of a general requirements document, to retain an overview of all requirements. This also makes it natural to tie security requirements to the relevant functional requirements, as Haley and his colleagues suggest.\(^8\)

**Does lightweight equal worthless?**

As we mentioned, we’re aware that a lightweight approach won’t be bullet-proof. However, if “normal” applications constitute, say, 90 percent of the global development effort, even a 5-percent reduction in security vulnerabilities will have a significant impact. Helping developers choose which techniques to use, and providing detailed, step-by-step guidance in applying these techniques, will bring us a step closer to the holy grail of secure software—if for no other reason than to make developers at least think about security requirements in their development projects.

No matter how formally pleasing or academically correct a method might be, if the intended audience isn’t using it, it isn’t good enough. Instead of creating yet another methodology, we’ve looked at what’s considered good
practice in security requirements engineering and selected a low-threshold set of techniques that should be palatable to the average software developer.

Altering development methodologies overnight is impossible, so a success criterion will be the education of software developers. Teaching old dogs new tricks is always difficult, so we advocate introducing secure software engineering as a mandatory curriculum component for all software engineering students. (This is quite different from the computer security courses that already are a part of many undergraduate computer science and computer engineering programs.) In this context especially, a lightweight approach is the only way to go: Unless techniques are easy to understand and to use, even impressionable students will shy away from them.

We plan to refine the ideas presented here into a full-fledged lightweight security requirement elicitation method with step-by-step guidance. We’ll then validate and improve this method by performing more practical studies—for example, we’ll use it in student development projects, software research projects, and industrial software development projects.

Acknowledgments

We thank Bashar Nuseibeh of the Open University, our Sintef colleagues, and the anonymous reviewers for their useful comments.

References

24. L. Rastad, “An Extended Misuse Case Notation: In-
CALL FOR ARTICLES

Opportunistic Software Systems Development

Increasing user demands and costs are driving the software development industry toward finding and gluing together pieces of software that were never designed to work together. Opportunistic software systems are not composed of homogeneous software pieces with clear boundaries and well-described functionalities. Few rules exist for design, construction, or process management. Practitioners need unique sets of skills, tools, and techniques as well as imagination and innovation to cope with these challenges and opportunities.

This issue aims to describe the state of the practice, broaden the software community’s knowledge of this systems development style, publicize core and critical issues that must still be addressed, and map out promising future directions. We welcome both supporting and opposing views.

Publication: November/December 2008
Submission deadline: 21 March 2008

Contributions might cover:
- underlying concepts;
- advantages and impediments;
- necessary skills;
- identification of appropriate software subsystems;
- requirements, design, development, maintenance, and evolution challenges;
- case studies, lessons learned, and experiences;
- suitable architectural styles and patterns;
- gluing strategies;
- supporting tools and technologies;
- conditions under which opportunistic development is appropriate;
- effects on life cycle; and
- implications on management and governance.

Guest Editors:
- Cornelius Ncube, Bournemouth University, cncube@bournemouth.ac.uk
- Patricia Oberndorf, Software Engineering Institute, po@sei.cmu.edu
- Anatol W. Kark, National Research Council Canada, Anatol.Kark@nrc-cnrc.gc.ca

For a full call for papers, see www.computer.org/software/cfp.htm or http://softwareresearch.ca/seg/OSSD. For IEEE Software author guidelines and submission details, visit www.computer.org/software/author.htm or contact the publications coordinator (software@computer.org). Submit your article via the Computer Society’s Electronic Submission System (http://mc.manuscriptcentral.com/cs-ieee), specifying it is for the “Opportunistic Software Systems Development” special issue.