SEI SERIES • A CERT® BOOK

SOFTWARE SECURITY SERIES

Software Security Engineering

A Guide for Project Managers

Julia H. Allen • Sean Barnum
Robert J. Ellison • Gary McGraw
Nancy R. Mead

Chapter 3

Requirements Engineering for Secure Software

3.1 Introduction

When security requirements are considered at all during the system life cycle, they tend to be general lists of security features such as password protection, firewalls, virus detection tools, and the like. These are, in fact, not security requirements at all, but rather implementation mechanisms that are intended to satisfy unstated requirements, such as authenticated access. As a result, security requirements that are specific to the system and that provide for protection of essential services and assets are often neglected. In addition, the attacker perspective is not considered, with the result being that security requirements—when they exist—are likely to be incomplete. We believe that a systematic approach to security requirements engineering will help avoid the problem of generic lists of features and take into account the attacker's perspective. Several approaches to security requirements engineering are described in this chapter, and references are provided to additional material that can help you ensure that your products effectively meet security requirements.

3.1.1 The Importance of Requirements Engineering

It comes as no surprise that requirements engineering is critical to the success of any major development project. Some studies have shown that requirements engineering defects cost 10 to 200 times as much to correct once the system has become operational than if they were detected during requirements development [Boehm 1988; McConnell 2001]. Other studies have shown that reworking requirements, design, and code defects on most software development projects accounts for 40 to 50 percent of the total project effort [Jones 1986a]; the percentage of defects originating during requirements engineering is estimated at more than 50 percent. The total percentage of project budget due to requirements defects ranges from 25 percent to 40 percent [Wiegers 2003]. Clearly, given these costs of poor security requirements, even a small improvement in this area would provide a high value. By the time that an application is installed in its operational environment, it is very difficult and expensive to significantly improve its security.

Requirements problems are among the top causes of the following undesirable phenomena [Charette 2005]:

- Projects are significantly over budget, go past schedule, have significantly reduced scope, or are cancelled
- Development teams deliver poor-quality applications
- Products are not significantly used once delivered

These days we have the further problem that the environment in which we do requirements engineering has changed, resulting in an added element of complexity. Today's software development takes place in a dynamic environment that changes while projects are still in development, with the result that requirements are constantly in a state of flux. Such changes can be inspired by a variety of causes—conflicts between stakeholder groups, rapidly evolving markets, the impact of tradeoff decisions, and so on.

In addition, requirements engineering on individual projects often suffers from the following problems:

- Requirements identification typically does not include all relevant stakeholders and does not use the most modern or efficient techniques.

- Requirements are often statements describing architectural constraints or implementation mechanisms rather than statements describing what the system must do.
- Requirements are often directly specified without any analysis or modeling. When analysis *is* done, it is usually restricted to functional end-user requirements, ignoring (1) quality requirements such as security, (2) other functional and nonfunctional requirements, and (3) architecture, design, implementation, and testing constraints.
- Requirements specification is typically haphazard, with specified requirements being ambiguous, incomplete (e.g., nonfunctional requirements are often missing), inconsistent, not cohesive, infeasible, obsolete, neither testable nor capable of being validated, and not usable by all of their intended audiences.
- Requirements management is typically weak, with ineffective forms of data capture (e.g., in one or more documents rather than in a database or tool) and missing attributes. It is often limited to tracing, scheduling, and prioritization, without change tracking or other configuration management. Alternatively, it may be limited to the capabilities provided by a specific tool, with little opportunity for improvement.

3.1.2 Quality Requirements

Even when organizations recognize the importance of functional end-user requirements, they often neglect *quality* requirements, such as performance, safety, security, reliability, and maintainability. Some quality requirements are nonfunctional requirements, but others describe system functionality, even though it may not contribute directly to end-user requirements.

As you might expect, developers of certain kinds of mission-critical systems and systems in which human life is involved, such as the space shuttle, have long recognized the importance of quality requirements and have accounted for them in software development. In many other systems, however, quality requirements are ignored altogether or treated in an inadequate way. Hence we see the failure of software associated with power systems, telephone systems, unmanned spacecraft, and so on. If quality requirements are not attended to in these types of systems, it is far less likely that they will be focused on in ordinary business systems.

This inattention to quality requirements is exacerbated by the desire to keep costs down and meet aggressive schedules. As a consequence, software development contracts often do not contain specific quality requirements, but rather offer up some vague generalities about quality, if they touch on this topic at all.

3.1.3 Security Requirements Engineering

If security requirements are not effectively defined, the resulting system cannot be evaluated for success or failure prior to its implementation [BSI 09]. When security requirements are considered, they are often developed independently of other requirements engineering activities. As a result, specific security requirements are often neglected, and functional requirements are specified in blissful ignorance of security aspects.

In reviewing requirements documents, we typically find that security requirements—when they exist—are in a section by themselves and have been copied from a generic list of security features. The requirements elicitation and analysis that are needed to produce a better set of security requirements seldom take place.

As noted previously, operational environments and business goals often change dynamically, with the result that security requirements development is not a one-time activity. Therefore the activities that we describe in this chapter should be planned as iterative activities, taking place as change occurs. Although we describe them as one-time activities for the sake of exposition, you can expect mini-life cycles to occur over the course of a project.

Much requirements engineering research and practice addresses the capabilities that the system will provide. As a consequence, a lot of attention is paid to the functionality of the system from the user's perspective, but little attention is devoted to what the system should *not* do [Bishop 2002]. Users have implicit assumptions for the software applications and systems that they use. They expect those products to be secure and are surprised when they are not. These user assumptions need to be translated into security requirements for the software systems when they are under development. Often the implicit assumptions of users are overlooked, and features are focused on instead.

Another important perspective is that of the attacker. An attacker is not particularly interested in functional features of the system, unless

they provide an avenue for attack. Instead, the attacker typically looks for defects and other conditions outside the norm that will allow a successful intrusion to take place. For this reason, it is important for requirements engineers to think about the attacker's perspective and not just the functionality of the system from the end-user's perspective. The discussion of attack patterns in Chapter 2 provides a good place to start this analysis. Other techniques that can be used in defining the attacker's perspective are misuse and abuse cases [McGraw 2006], attack trees [Ellison 2003; Schneier 2000], and threat modeling [Howard 2002]. Some of these methodologies are discussed in later sections of this chapter.

For many projects, security requirements are stated as negative requirements. As a result, general security requirements, such as "The system shall not allow successful attacks," are usually not feasible, as there is no consensus on ways to validate them other than to apply formal methods to the entire system. We can, however, identify the essential services and assets that must be protected. Operational usage scenarios can be extremely helpful aids to understanding which services and assets are essential. By providing threads that trace through the system, such scenarios also help to highlight security requirements as well as other quality requirements such as safety and performance [Reifer 2003]. Once the essential services and assets are understood, we become able to validate that mechanisms such as access control, levels of security, backups, replication, and policy are implemented and enforced. We can also validate that the system properly handles specific threats identified by a threat model and correctly responds to intrusion scenarios.

As usable approaches to security requirements engineering continue to emerge and new mechanisms are identified to promote organizational use, project managers can do a better job of ensuring that the resulting product effectively meets security requirements. The following techniques are known to be useful in this regard:

- Comprehensive, Lightweight Application Security Process (CLASP) approach to security requirements engineering. CLASP is a lifecycle process that suggests a number of different activities across the development life cycle in an attempt to improve security. Among these is a specific approach for security requirements [BSI 12].
- Security Quality Requirements Engineering (SQUARE). This process is aimed specifically at security requirements engineering.

- Core security requirements artifacts [Moffett 2004]. This approach takes an artifact view and starts with the artifacts that are needed to achieve better security requirements. It provides a framework that includes both traditional requirements engineering approaches to functional requirements and an approach to security requirements engineering that focuses on assets and harm to those assets.

Other useful techniques include formal specification approaches to security requirements, such as Software Cost Reduction (SCR) [Heitmeyer 2002], and the higher levels of the Common Criteria [CCMB 2005a]. As an additional reference, the SOAR report *Software Security Assurance* [Goertzel 2007] contains a good discussion of SDLC processes and various approaches to security requirements engineering.

In this chapter we discuss several approaches to development of security requirements, including the use of misuse and abuse cases, security quality requirements engineering, security requirements elicitation, and security requirements prioritization. While the processes we discuss are similar to those used for requirements engineering in general, we have found that when we delve into the detailed steps of how to do security requirements engineering, certain techniques are particularly useful, and we highlight these where they occur.

3.2 Misuse and Abuse Cases[1]

L L3

To create secure and reliable software, we first must anticipate abnormal behavior. We don't normally describe non-normative behavior in use cases, nor do we describe it with UML, but we must have some way to talk about and prepare for it. *Misuse* (or *abuse*) cases can help you begin to see your software in the same light that attackers do. By thinking beyond normative features while simultaneously contemplating negative or unexpected events, you can better understand how to create secure and reliable software.[2]

1. [BSI 43] © 2004 IEEE. Reprinted, with permission, from "Misuse and Abuse Cases: Getting Past the Positive" by Paco Hope, Gary McGraw, and Annie I. Anton, *IEEE Security & Privacy* 2, 3 (May/June 2004): 90–92.

2. Since the original publication of this material, there have been a number of vendor efforts to improve security, such as the Microsoft effort described in [Howard 2007].

Guttorm Sindre and Andreas Opdahl extend use-case diagrams with misuse cases to represent the actions that systems should prevent in tandem with those that they should support for security and privacy requirements analysis [Sindre 2000]. Ian Alexander advocates using misuse and use cases together to conduct threat and hazard analysis during requirements analysis [Alexander 2003]. Here, we provide a nonacademic introduction to the software security best practice of misuse and abuse cases, showing you how to put the basic science to work.

3.2.1 Security Is Not a Set of Features

There is no convenient security pull-down menu that will let you select "security" and then sit back and watch magic things happen. Unfortunately, many software developers simply link functional security features and mechanisms somewhere into their software, mistakenly assuming that doing so addresses security needs throughout the system. Too often, product literature makes broad, feature-based claims about security, such as "built with SSL" or "128-bit encryption included," which represent the vendor's entire approach for securing its product.

Security is an emergent property of a system, not a feature. This is analogous to how "being dry" is an emergent property of being inside a tent in the rain. The tent will keep you dry only if the poles are stabilized, vertical, and able to support the weight of wet fabric; the tent also must have waterproof fabric (with no holes) and be large enough to protect everyone who wants to remain dry. Lastly, everyone must remain under the tent for the entire time it's raining. So, although having poles and fabric is important, it's not enough to say, "The tent has poles and fabric; thus it keeps you dry!" This sort of claim, however, is analogous to the claims that software vendors make when they highlight numbers of bits in cryptographic keys or the use of particular encryption algorithms. Cryptography of one kind or another is usually necessary to create a secure system, but security features alone are not sufficient for building secure software.

Because security is not a feature, it cannot be bolted on after other software features are codified, nor can it be patched in after attacks have occurred in the field. Instead, security must be built into the product from the ground up, as a critical part of the design from the very beginning (requirements specification) and included in every subsequent development phase, all the way through fielding a complete system.

Sometimes building security in at the beginning of the SDLC means making explicit tradeoffs when specifying system requirements. For example, ease of use might be paramount in a medical system designed for clerical personnel in doctors' offices, but complex authentication procedures, such as obtaining and using a cryptographic identity, can be hard to use [Whitten 1999]. Furthermore, regulatory pressures from HIPAA and California's privacy regulations (Senate Bill 1386) force designers to negotiate a reasonable tradeoff.

Technical approaches must go far beyond the obvious features, deep into the many-tiered heart of a software system, to provide enough security: Authentication and authorization can't stop at a program's front door. The best, most cost-effective approach to software security incorporates thinking beyond normative features and maintains that thinking throughout the development process. Every time a new requirement, feature, or use case is created, the developer or security specialist should spend some time thinking about how that feature might be unintentionally misused or intentionally abused. Professionals who know how features are attacked and how to protect software should play active roles in this kind of analysis.

3.2.2 Thinking About What You Can't Do

Attackers are not standard-issue customers. They're bad people with malicious intentions who want your software to act to their benefit. If the development process doesn't address unexpected or abnormal behavior, then an attacker usually has plenty of raw material with which to work [Hoglund 2004].

Although attackers are creative, they always probe well-known locations—boundary conditions, edges, intersystem communication, and system assumptions—in the course of their attacks. Clever attackers will try to undermine the assumptions on which a system was built. If a design assumes that connections from the Web server to the database server are always valid, for example, an attacker will try to make the Web server send inappropriate requests to access valuable data. If the software design assumes that the client never modifies its Web browser cookies before they are sent back to the requesting server (in an attempt to preserve some state), attackers will intentionally cause problems by modifying the cookies. *Building Secure Software* teaches us that we have to be on guard when we make any assumptions [Viega 2001].

When we design and analyze a system, we're in a great position to know our systems better than potential attackers do. We must leverage this knowledge to the benefit of security and reliability, which we can achieve by asking and answering the following critical questions: Which assumptions are implicit in our system? Which kinds of things make our assumptions false? Which kinds of attack patterns will an attacker bring to bear?

Unfortunately, a system's creators are not the best security analysts of that system. Consciously noting and considering all assumptions (especially in light of thinking like an attacker) is extremely difficult for those who have built up a set of implicit assumptions. Fortunately, these professionals make excellent subject matter experts (SMEs). Together, SMEs and security analysts can ferret out base assumptions in a system under analysis and think through the ways an attacker will approach the software.

3.2.3 Creating Useful Misuse Cases

One of the goals of misuse cases is to decide and document *a priori* how software should react to illegitimate use. The simplest, most practical method for creating misuse cases is usually through a process of informed brainstorming. Several theoretical methods require fully specifying a system with rigorous formal models and logics, but such activities are extremely time and resource intensive. A more practical approach teams security and reliability experts with SMEs. This approach relies heavily on expertise and covers a lot of ground quickly.

To guide brainstorming, software security experts ask many questions of a system's designers to help identify the places where the system is likely to have weaknesses. This activity mirrors the way attackers think. Such brainstorming involves a careful look at all user interfaces (including environmental factors) and considers events that developers assume a person can't or won't do. These "can'ts" and "won'ts" take many forms: "Users can't enter more than 50 characters because the JavaScript code won't let them" or "Users don't understand the format of the cached data, so they can't modify it." Attackers, unfortunately, can make these can'ts and won'ts happen.

The process of specifying abuse cases makes a designer very clearly differentiate appropriate use from inappropriate use. To reach this point, however, the designer must ask the right questions: How can the system distinguish between good input and bad input? Can it tell whether a

request is coming from a legitimate application or from a rogue application replaying traffic? All systems have more vulnerable places than the obvious front doors, so where might a bad guy be positioned? On the wire? At a workstation? In the back office? Any communication line between two endpoints or two components is a place where an attacker might try to interpose himself or herself, so what can this attacker do in the system? Watch communications traffic? Modify and replay such traffic? Read files stored on the workstation? Change registry keys or configuration files? Be the DLL? Be the "chip"?

Trying to answer such questions helps software designers explicitly question design and architecture assumptions, and it puts the designer squarely ahead of the attacker by identifying and fixing a problem before it's ever created.

3.2.4 An Abuse Case Example

This section describes a real-world example of a classic software security problem on a client/server application. The architecture had been set up so that the server relied on the client-side application, which manipulated a financially sensitive database, to manage all data-access permissions—no permissions were enforced on the server itself. In fact, only the client had any notion of permissions and access control. To make matters worse, a complete copy of the database (only parts of which were to be viewed by a given user with a particular client) was sent to the client program, which ran on a garden-variety desktop PC. As a consequence, a complete copy of the sensitive data (which was expressly *not* to be viewed by the user) was available on that user's PC in the clear. If the user looked in the application's cache on the hard disk and used a standard-issue unzip utility, he or she could see all sorts of sensitive information.

The client also enforced which messages were sent to the server, honoring these messages independent of the user's actual credentials. The server assumed that any messages coming from the client had passed the client software's access control system (and policy) and were, therefore, legitimate. By intercepting network traffic, corrupting values in the client software's cache, or building a hostile client, malicious users could inject data into the database that they were not even supposed to read (much less write to).

Determining the can'ts and won'ts in such a case is difficult for those who think only about positive features. Attack patterns can provide

Misuse/Abuse Case Templates

Templates for misuse and abuse cases appear in a number of references. They can be text or diagrams, and some are supported by tools. Some good sources for templates are in materials by Sindre and Opdahl [Sindre 2001] and Alexander [Alexander 2002]. Figure 3–1 is an example of a use/misuse-case diagram to elicit security requirements from Alexander's article. The high-level case is shown on the left; use cases are drawn in white and misuse cases are drawn in black.

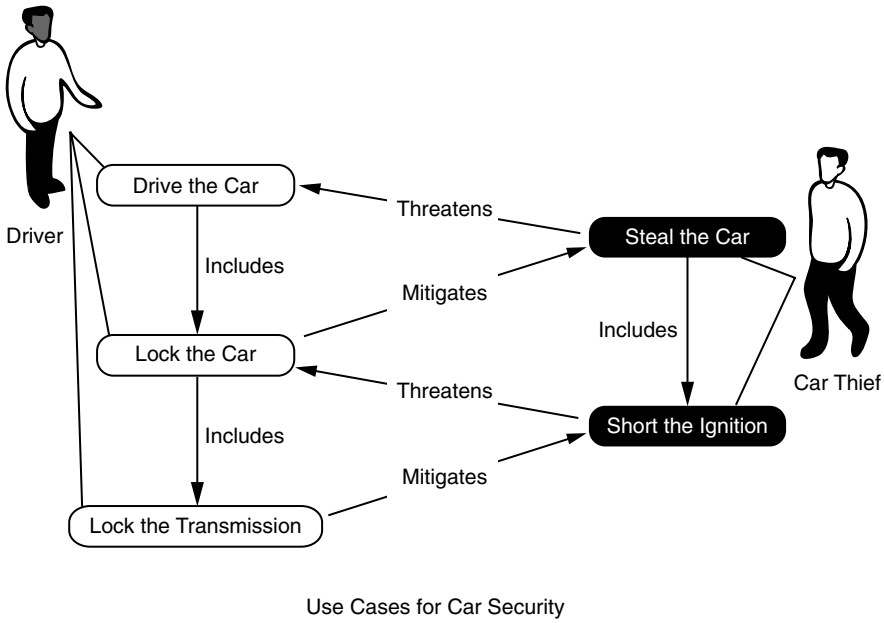

Figure 3–1: *Misuse case example*

some guidance in this regard (see Section 2.3.2). Attack patterns are akin to patterns in sewing—that is, a blueprint for creating an attack. Everyone's favorite example, the buffer overflow, follows several different standard patterns, but patterns allow for a fair amount of variation on a theme. They can take into account many dimensions, including timing, resources required, techniques, and so forth [Hoglund 2004]. When we're trying to develop misuse and abuse cases, attack patterns can help.

It is possible for misuse cases to be overused (and generated forever with little impact on actual security). A solid approach to building them requires a combination of security know-how and subject matter expertise to prioritize misuse cases as they are generated and to strike the right balance between cost and value.

Although misuse and abuse cases can be used as a stand-alone activity, they are more effective when they are developed as part of an overall security requirements engineering process. As noted in Section 3.1.3, a number of processes can be used to address security requirements engineering. In the next section, we describe one such process, the SQUARE process model, in which misuse and abuse cases play important roles. Consult the reference material that we have provided to learn about other processes and select the process and methods that are best for your organization.

3.3 The SQUARE Process Model

Security Quality Requirements Engineering (SQUARE) is a process model that was developed at Carnegie Mellon University [Mead 2005].[3] It provides a means for eliciting, categorizing, and prioritizing security requirements for information technology systems and applications. (Note that this section and the following sections all discuss security requirements, regardless of whether the term "security" is specifically used as a qualifier.) The focus of the model is to build security concepts into the early stages of the SDLC. It can also be used for documenting and analyzing the security aspects of systems once they are implemented in the field and for steering future improvements and modifications to those systems.

After its initial development, SQUARE was applied in a series of client case studies [Chen 2004; Gordon 2005; Xie 2004]. Prototype tools were also developed to support the process. The draft process was revised and established as a baseline after the case studies were completed; the baselined process is shown in Table 3–1. In principle, Steps 1–4 are actually activities that precede security requirements engineering but are necessary to ensure that it is successful. Brief descriptions of each step follow; a detailed discussion of the method can be found in [Mead 2005].

3. The SQUARE work is supported by the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab.

Table 3–1: *The SQUARE Process*

Number	*Step*	*Input*	*Techniques*	*Participants*	*Output*
1	Agree on definitions	Candidate definitions from IEEE and other standards	Structured interviews, focus group	Stakeholders, requirements engineers	Agreed-to definitions
2	Identify security goals	Definitions, candidate goals, business drivers, policies and procedures, examples	Facilitated work session, surveys, interviews	Stakeholders, requirements engineers	Goals
3	Develop artifacts to support security requirements definition	Potential artifacts (e.g., scenarios, misuse cases, templates, forms)	Work session	Requirements engineers	Needed artifacts: scenarios, misuse cases, models, templates, forms
4	Perform (security) risk assessment	Misuse cases, scenarios, security goals	Risk assessment method, analysis of anticipated risk against organizational risk tolerance, including threat analysis	Requirements engineers, risk expert, stakeholders	Risk assessment results

Continues

Table 3–1: *The SQUARE Process (Continued)*

Number	*Step*	*Input*	*Techniques*	*Participants*	*Output*
5	Select elicitation techniques	Goals, definitions, candidate techniques, expertise of stakeholders, organizational style, culture, level of security needed, cost–benefit analysis	Work session	Requirements engineers	Selected elicitation techniques
6	Elicit security requirements	Artifacts, risk assessment results, selected techniques	Accelerated Requirements Method, Joint Application Development, interviews, surveys, model-based analysis, checklists, lists of reusable requirements types, document reviews	Stakeholders facilitated by requirements engineers	Initial cut at security requirements

Table 3–1: *The SQUARE Process (Continued)*

Number	*Step*	*Input*	*Techniques*	*Participants*	*Output*
7	Categorize requirements as to level (e.g., system, software) and whether they are requirements or other kinds of constraints	Initial requirements, architecture	Work session using a standard set of categories	Requirements engineers, other specialists as needed	Categorized requirements
8	Prioritize requirements	Categorized requirements and risk assessment results	Prioritization methods such as Analytical Hierarchy Process (AHP), triage, and win-win	Stakeholders facilitated by requirements engineers	Prioritized requirements
9	Inspect requirements	Prioritized requirements, candidate formal inspection technique	Inspection method such as Fagan and peer reviews	Inspection team	Initial selected requirements, documentation of decision-making process and rationale

3.3.1 A Brief Description of SQUARE

The SQUARE process is best applied by the project's requirements engineers and security experts in the context of supportive executive management and stakeholders. We have observed that this process works best when elicitation occurs after risk assessment (Step 4) has been done and when security requirements are specified before critical architecture and design decisions. Thus critical security risks to the business will be considered in the development of the security requirements.

Step 1, "Agree on definitions," is needed as a prerequisite to security requirements engineering. On a given project, team members tend to have definitions in mind, based on their prior experience, but those definitions often differ [Woody 2005]. For example, for some government organizations, security has to do with access based on security clearance levels, whereas for others security may have to do with physical security or cybersecurity. It is not necessary to invent definitions. Sources such as the Institute for Electrical and Electronics Engineers (IEEE) and the Software Engineering Body of Knowledge (SWEBOK) provide a range of definitions to select from or tailor. A focus group meeting with the interested parties will most likely enable the selection of a consistent set of definitions for the security requirements activity.

Step 2, "Identify security goals," should be done at the organizational level and is needed to support software development in the project at hand. This step provides a consistency check with the organization's policies and operational security environment. Different stakeholders usually have different goals. For example, a stakeholder in human resources may be concerned about maintaining the confidentiality of personnel records, whereas a stakeholder in a financial area may be concerned with ensuring that financial data is not accessed or modified without authorization. It is important to have a representative set of stakeholders, including those with operational expertise. Once the goals of the various stakeholders have been identified, they need to be prioritized. In the absence of consensus, an executive decision may be needed to prioritize them. It is expected that the goals identified in this step will link to the core properties discussed in Chapter 2.

Step 3, "Develop artifacts," is necessary to support all subsequent security requirements engineering activities. Organizations often do not have a documented concept of operations for a project, succinctly

stated project goals, documented normal usage and threat scenarios, misuse or abuse cases, and other documents needed to support requirements definition. As a consequence, either the entire requirements process is built on unstated assumptions or a lot of time is spent backtracking to try to obtain such documentation.

Step 4, "Perform risk assessment," requires an expert in risk assessment methods, the support of the stakeholders, and the support of a security requirements engineer. A number of risk assessment methods are available (as discussed in detail in Section 3.4.1). The risk assessment expert can recommend a specific method based on the unique needs of the organization. The artifacts from Step 3 provide the input to the risk assessment process; the outcomes of the risk assessment, in turn, can help in identifying the high-priority security exposures (see also the discussion of the risk management framework in Section 7.4). Organizations that do not perform risk assessment typically do not have a logical approach to considering organizational risks when identifying security requirements, but rather tend to select specific solutions or technologies, such as encryption, without really understanding the problem that is being solved.

Step 5, "Select elicitation technique," becomes important when the project has diverse stakeholders. A more formal elicitation technique, such as the Accelerated Requirements Method [Hubbard 1999], Joint Application Design [Wood 1989], or structured interviews, can be effective in overcoming communication issues when stakeholders have variable cultural backgrounds. In other cases, elicitation may simply consist of sitting down with a primary stakeholder and trying to understand that stakeholder's security requirements needs.

Step 6, "Elicit security requirements," is the actual elicitation process using the selected technique. Most elicitation techniques provide detailed guidance on how to perform elicitation. This effort builds on the artifacts that were developed in earlier steps, such as misuse and abuse cases, attack trees, threats, and scenarios.

Step 7, "Categorize requirements," allows the security requirements engineer to distinguish among essential requirements, goals (desired requirements), and architectural constraints that may be present. Requirements that are actually constraints typically arise when a specific system architecture has been chosen prior to the requirements process. This is good, as it allows for assessment of the risks associated with these constraints. This categorization also helps in the prioritization activity that follows (Step 8).

Step 8, "Prioritize requirements," depends not only on the prior step, but may also involve performing a cost–benefit analysis to determine which security requirements have a high payoff relative to their cost. Prioritization may also depend on other consequences of security breaches, such as loss of life, loss of reputation, and loss of consumer confidence.

Step 9, "Requirements inspection," can be done at varying levels of formality, ranging from Fagan inspections (a highly structured and proven technique for requirements inspection) [Fagan 1999] to peer reviews. Once this inspection is complete, the project team should have an initial set of prioritized security requirements. It should also understand which areas are incomplete and must be revisited at a later time. Finally, the project team should understand which areas are dependent on specific architectures and implementations and should plan to revisit those areas as well.

3.3.2 Tools

A prototype tool has been developed to support SQUARE. It primarily provides an organizational framework for the artifact documents; in addition, it provides default content for some of the steps. The tool does not perform sophisticated functions such as requirements analysis. This prototype is undergoing further development so that it will provide better support to the SQUARE process and be more attractive to users. The current status of the SQUARE process and tool, as well as contact information, can be found at http://www.cert.org/nav/index_purple.html/square.html.

3.3.3 Expected Results

When you apply SQUARE, you can expect relevant security requirements to be identified and documented for the system or software that is being developed. SQUARE is better suited to use with a system under development than with a system that has already been fielded, although it has been used in both situations. Although quantitative measures do not exist, case study clients recognized the value of the new security requirements and have taken steps to incorporate them into their system specifications. You'll need to consider the resources required for this activity and for the implementation of the resulting requirements [Xie 2004].

Our experience with SQUARE suggests that the system and its elements must be considered within the context or environment in which it operates. For example, a system that operates on a single isolated workstation will have very different security requirements from a similar system that is Web based. Likewise, a medical information system will have different security requirements for workstations that are isolated in a physician's office than for those that are located in a public area in a hospital. These differences should be accounted for in the artifacts developed in Step 3—for example, in usage scenarios and misuse or abuse cases. When the context for a project changes, you should revisit the security requirements and reapply the SQUARE process. It may be that a subset of the SQUARE steps will be sufficient for this purpose, but we do not yet have enough experience with subsequent applications of SQUARE to the same system to make that determination.

3.4 SQUARE Sample Outputs

Several case studies have been conducted using the SQUARE process model [Chen 2004; Gordon 2005]. The goals of these case studies were to experiment with each step of the SQUARE process, make recommendations, and determine the feasibility of integrating the SQUARE methodology into standard software development practices. The case studies involved real-world clients that were developing large-scale IT projects, including an IT firm in Pittsburgh, Pennsylvania; a federal government research institute; and a department of the federal government.

Acme Corporation (an alias used to protect the identity of the client), a private IT firm headquartered in Pittsburgh, provides technical and management services to various public sectors and a number of diversified private sectors. Its product, the Asset Management System (AMS) version 2, provides a tool that enables companies to make strategic allocations and plans for their critical IT assets. This system provides specialized decision support capabilities via customized views. AMS provides a graphical interface to track and analyze the state of important assets. The security requirements surrounding the AMS are the subject of one of our case studies and the source of the sample outputs that follow.

3.4.1 Output from SQUARE Steps

We present a sample output for each step, all taken from the case studies, to provide concrete examples of the nine SQUARE steps. Given that these are actual results, they are not all that sophisticated or cutting edge, but they do reflect the typical state of affairs at present. Note, however, that these snippets leave out underlying assumptions and background information.

Step 1: Agree on Definitions

We worked with the client to agree on a common set of security definitions with which to create a common base of understanding. The following is a small subset of the definitions that were agreed to:

- *Access control:* Ensures that resources are granted only to those users who are entitled to them.
- *Access control list:* A table that tells a computer operating system which access rights or explicit denials each user has to a particular system object, such as a file directory or individual file.
- *Antivirus software:* A class of program that searches hard drives and memory for any known or potential viruses.

The full set of definitions was drawn from resources such as IEEE, Carnegie Mellon University, industry, and various dictionaries.

Step 2: Identify Security Goals

We worked with the client to flesh out security goals that mapped to the company's overall business goals. This is one example set of goals:

- *Business goal of AMS:* To provide an application that supports asset management and planning.
- *Security goals:* Three high-level security goals were derived for the system (it's not surprising that these are closely linked to the security properties of Chapter 2):
  a. Management shall exercise effective control over the system's configuration and use.
  b. The confidentiality, accuracy, and integrity of the AMS shall be maintained.
  c. The AMS shall be available for use when needed.

Step 3: Develop Artifacts

Architectural diagrams, use cases, misuse cases, abuse case diagrams, attack trees, and essential assets and services were documented in this step. As noted earlier, the attack patterns discussed in Chapter 2 provide a good starting point for developing artifacts that reflect the attacker's perspective. For instance, an attack scenario was documented in the following way:

System administrator accesses confidential information

1. by being recruited OR
   a. by being bribed OR
   b. by being threatened OR
   c. through social engineering OR
2. by purposefully abusing rights

An example abuse case diagram is shown in Figure 3–2.

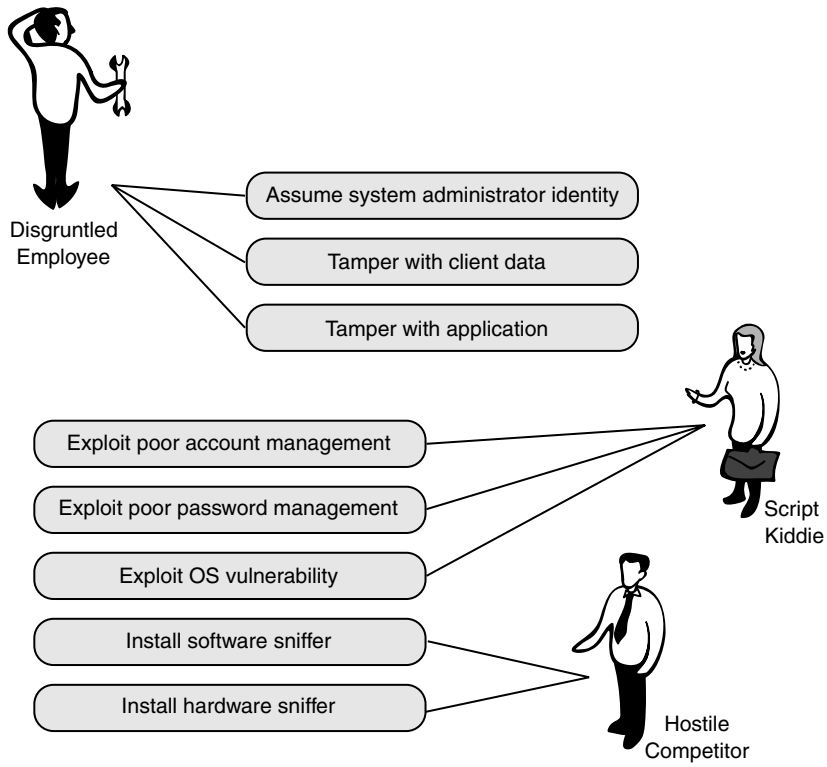

Figure 3–2: *Abuse case example*

This step creates needed documentation that serves as input for the following steps.

Step 4: Perform Risk Assessment

The risk assessment techniques that were field tested were selected after completing a literature review. This review examined the usefulness and applicability of eight risk assessment techniques:

1. General Accounting Office Model [GAO 1999]
2. National Institute of Standards and Technology (NIST) Model [Stoneburner 2002]
3. NSA's INFOSEC Assessment Methodology [NSA 2004]
4. Shawn Butler's Security Attribute Evaluation Method [Butler 2002]
5. Carnegie Mellon's Vendor Risk Assessment and Threat Evaluation [Lipson 2001]
6. Yacov Haimes's Risk Filtering, Ranking, and Management Model [Haimes 2004]
7. Carnegie Mellon's Survivable Systems Analysis Method [Mead 2002]
8. Martin Feather's Defect Detection and Prevention Model [Cornford 2004]

Each method was ranked in four categories:

1. Suitability for small companies
2. Feasibility of completion in the time allotted
3. Lack of dependence on historical threat data
4. Suitability in addressing requirements

The results of the ranking are shown in Table 3–2.

After averaging scores from the four categories, NIST's and Haimes's models were selected as useful techniques for the risk assessment step. Brainstorming, attack tree, and misuse case documentation were used to identify potential threat scenarios. The two independent risk assessment analyses produced a useful risk profile for the company's system, with two especially meaningful findings:

- Insider threat poses the highest-impact risk to the AMS.
- Because of weak controls, it is easy for an insider or unauthorized user to defeat authentication.

Table 3–2: *Ranking of Assessment Techniques*

		Suitable for Small Companies	*Feasible to Complete within Time Frame*	*Does Not Require Additional Data Collection*	*Suitable for Requirements*	*Average Score*
Methodologies	GAO	2	4	2	2	2.50
	NIST	2	2	1	1	1.50
	NSA/IAM	3	3	2	2	2.50
	SAEM	4	4	4	4	4.00
	V-Rate	3	4	4	4	3.75
	Haimes	2	2	2	2	2.00
	SSA	2	2	2	4	2.50
	DDP/Feather	3	4	2	4	3.25

In this particular case study, we also identified a set of essential services and assets as part of the artifact generation. This activity is not part of the standard SQUARE process but nevertheless can be a beneficial exercise if enough architectural information already exists to support it. All findings from the risk assessment, along with the findings from the essential services and asset identification process, were used to determine the priority level associated with each of the nine requirements.

We analyzed the importance of each of the major system services, outlined in the 11 use cases shown in Table 3–3, and made a determination as to which were essential.

Table 3–3: *Classification of Use Cases*

Use Case	*Service*	*Status*
UC-1	View floor plans	***Essential***
UC-2	Enter damage assessment	***Essential***
UC-3	Add/delete/edit Post-it notes	Nonessential
UC-4	Find specialized employees	***Important***
UC-5	Create journal entry	Nonessential
UC-6	Install the Asset Management System	Nonessential
UC-7	Create links to documents	Nonessential
UC-8	Archibus administration: Add user and assign privileges	Nonessential
UC-9	View contact information for maintenance tasks	***Important***
UC-10	Create open space report	***Essential***
UC-11	View incident command	***Essential***

There are two essential assets in this system. The first is the Windows Server computer, which houses the majority of the production system's intellectual assets (that is, the code that runs the system). This computer acts as a server that allows remote users to access the Asset Management System. The second essential asset is the information inside the Windows Server computer—specifically, the files stored in the Microsoft IIS server and the information stored in the Sybase database and MapGuide database are critical for making informed decisions. If this information is lost or compromised, the ability to make accurate decisions is lost.

Step 5: Select Elicitation Techniques

For this step, teams tested various elicitation techniques and models. It is often the case that multiple techniques will work for the same project. The difficulty lies in choosing a technique that can be adapted to the number and expertise of stakeholders, the size and scope of the client project, and the expertise of the requirements engineering team. It is extremely unlikely that any single technique will work for all projects under all circumstances, although our experience has shown that the Accelerated Requirements Method [Hubbard 2000] has been successful in eliciting security requirements. Selection of an elicitation technique is discussed in more detail in Section 3.5.

Steps 6 and 7: Elicit and Categorize Security Requirements

Nine security requirements were identified and then organized to map to the three high-level security goals (see Step 2). Examples include the following requirements:

- Requirement 1: The system is required to have strong authentication measures in place at all system gateways and entrance points (maps to Goals 1 and 2).
- Requirement 2: The system is required to have sufficient process-centric and logical means to govern which system elements (e.g., data, functionality) users can view, modify, and/or interact with (maps to Goals 1 and 2).
- Requirement 3: A continuity of operations plan (COOP) is required to assure system availability (maps to Goal 3).
- Requirement 6: It is required that the system's network communications be protected from unauthorized information gathering and/or eavesdropping (maps to Goals 1 and 2).

The nine security requirements were central to the security requirements document that was ultimately delivered to the client.

Step 8: Prioritize Requirements

In the first case study, the nine security requirements were prioritized based on the following qualitative rankings:

- *Essential:* The product will be unacceptable if this requirement is absent.
- *Conditional:* The requirement enhances security, but the product is acceptable if this requirement is absent.
- *Optional:* The requirement is clearly of lower priority than essential and conditional requirements.

Requirement 1 from Steps 6 and 7, which dealt with authentication at borders and gateways, was deemed essential because of its importance in protecting against the high-impact, authentication-related risks identified in the risk assessment. Requirement 3, dealing with continuity of operations planning, was still seen as an important element and worth considering, but it was found to be an optional requirement relative to the other eight requirements. That is, although COOP plans are valuable, the risk assessment phase found that greater threats to the system resulted from unauthorized disclosure of information than from availability attacks.

We also used the Analytical Hierarchy Process (AHP) methodology to prioritize requirements and found it to be successful both in client acceptance and in its ability to handle security requirements [Karlsson 1997; Saaty 1980]. Requirements prioritization is discussed in more detail in Section 3.6.

Step 9: Requirements Inspection

We experimented with different inspection techniques and had varying levels of success with each. None of the inspection techniques was sufficiently effective in identifying defects in the security requirements. Instead, we recommend experimenting with the Fagan inspection technique.

In one case study instance, each team member played a role in inspecting the quality of the team's work and deliverables. A peer review log was created to document what had been reviewed and

was used to maintain a log of all problems, defects, and concerns. Each entry in the log was numbered and dated, and indicated the date, origin, defect type, description, severity, owner, reviewer, and status of the issue. Each entry was assigned to an owner, who was responsible for making sure that defects were fixed. This step was used as a sanity check to ensure that the system met quality goals and expectations.

3.4.2 SQUARE Final Results

The final output to the client was a security requirements document. The client could then use this document in the early stages of the SDLC to ensure that security requirements were built into project plans.

Once a system has been deployed, the organization can look back to its requirements documentation to analyze whether it met its requirements and, therefore, satisfied its security goals. As change occurs—be it a configuration concern in the system, the organization's risk profile, or a business goal—the SQUARE process can be revisited to determine how the change might affect the system's security requirements. In this way, SQUARE can be reapplied to a system as needed.

3.5 Requirements Elicitation

Using an elicitation method can help in producing a consistent and complete set of security requirements. However, brainstorming and elicitation methods used for ordinary functional (end-user) requirements usually are not oriented toward security requirements and, therefore, do not result in a consistent and complete set of security requirements. The resulting system is likely to have fewer security exposures when requirements are elicited in a systematic way.

In this section, we briefly discuss a number of elicitation methods and the kind of tradeoff analysis that can be done to select a suitable one. Companion case studies can be found in "Requirements Elicitation Case Studies" [BSI 10]. While results may vary from one organization to another, the discussion of our selection process and various methods should be of general utility. Requirements elicitation is an active research area, and we expect to see advances in this area in the future.

Eventually, studies will likely determine which methods are most effective for eliciting security requirements. At present, however, there is little if any data comparing the effectiveness of different methods for eliciting security requirements.

3.5.1 Overview of Several Elicitation Methods

The following list identifies several methods that could be considered for eliciting security requirements. Some have been developed specifically with security in mind (e.g., misuse cases), whereas others have been used for traditional requirements engineering and could potentially be extended to security requirements. In the future, we may have a better understanding of how the unique aspects of security requirements elicitation drive selection of a method. We also note recent work on requirements elicitation in general that could be considered in developing such a list [Hickey 2003, 2004; Zowghi 2005] and in doing the selection process [Hickey 2004]. We briefly describe each of the following elicitation methods:

- Misuse cases [Sindre 2000; McGraw 2006, pp. 205–222]
- Soft Systems Methodology [Checkland 1990]
- Quality Function Deployment [QFD 2005]
- Controlled Requirements Expression [Christel 1992; SDS 1985]
- Issue-based information systems [Kunz 1970]
- Joint Application Development [Wood 1995]
- Feature-oriented domain analysis [Kang 1990]
- Critical discourse analysis [Schiffrin 1994]
- Accelerated Requirements Method [Hubbard 2000]

Misuse Cases

As noted earlier, misuse/abuse cases apply the concept of a negative scenario—that is, a situation that the system's owner does *not* want to occur—in a use-case context. For example, business leaders, military planners, and game players are familiar with the strategy of analyzing their opponents' best moves as identifiable threats.

By contrast, a use case generally describes behavior that the system owner *wants* the system to show [Sindre 2000]. Use-case models and their associated diagrams (UCDs) have proven quite helpful for the

specification of requirements [Jacobson 1992; Rumbaugh 1994]. However, a collection of use cases should not be used as a substitute for a requirements specification document, as this approach can result in overlooking significant requirements [Anton 2001]. As a result, it is controversial to use only use-case models for system and quality requirements elicitation.

Soft Systems Methodology (SSM)

SSM deals with problem situations in which there is a high social, political, and human activity component [Checkland 1990]. The SSM can deal with "soft problems" that are difficult to define, rather than "hard problems" that are more technology oriented. Examples of soft problems include how to deal with homelessness, how to manage disaster planning, and how to improve Medicare. Eventually technology-oriented problems may emerge from these soft problems, but much more analysis is needed to reach that point.

The primary benefit of SSM is that it provides structure to soft problem situations and enables their resolution in an organized manner. In addition, it compels the developer to discover a solution that goes beyond technology.

Quality Function Deployment (QFD)

QFD is "an overall concept that provides a means of translating customer requirements into the appropriate technical requirements for each stage of product development and production" [QFD 2005]. The distinguishing attribute of QFD is the focus on customer needs throughout all product development activities. By using QFD, organizations can promote teamwork, prioritize action items, define clear objectives, and reduce development time [QFD 2005].

Controlled Requirements Expression (CORE)

CORE is a requirements analysis and specification method that clarifies the user's view of the services to be supplied by the proposed system. In CORE, the requirements specification is created by both the user and the developer—not solely one or the other. The problem to be analyzed is defined and broken down into user and developer viewpoints. Information about the combined set of viewpoints is then analyzed. The last step of CORE deals with constraints analysis, such as the limitations imposed by the system's operational

environment, in conjunction with some degree of performance and reliability investigation.

Issue-Based Information Systems (IBIS)

Developed by Horst Rittel, the IBIS method is based on the principle that the design process for complex problems, which Rittel terms *wicked* problems, is essentially an exchange among the stakeholders in which each stakeholder brings his or her personal expertise and perspective to the resolution of design issues [Kunz 1970]. Any problem, concern, or question can be an issue and may require discussion and resolution for the design to proceed.

Joint Application Development (JAD)

The JAD methodology [Wood 1995] is specifically designed for the development of large computer systems. Its goal is to involve all stakeholders in the design phase of the product via highly structured and focused meetings. In the preliminary phases of JAD, the requirements engineering team is charged with fact-finding and information-gathering tasks. Typically, the outputs of this phase, as applied to security requirements elicitation, are security goals and artifacts. The actual JAD session is then used to validate this information by establishing an agreed-on set of security requirements for the product.

Feature-Oriented Domain Analysis (FODA)

FODA is a domain analysis and engineering method that focuses on developing reusable assets [Kang 1990]. By examining related software systems and the underlying theory of the class of systems they represent, domain analysis can provide a generic description of the requirements of that class of systems in the form of a domain model and a set of approaches for their implementation.

The FODA method was founded on two modeling concepts: abstraction and refinement [Kean 1997]. Abstraction is used to create domain models from the specific applications in the domain. Specific applications in the domain are then developed as refinements of the domain models. The example domain used in the initial report on FODA [Kang 1990] is window management systems. The window management examples of that time are no longer in use, but include VMS, Sun, and Macintosh, among others.

Critical Discourse Analysis (CDA)

CDA uses sociolinguistic methods to analyze verbal and written discourse [Schiffrin 1994]. In particular, this technique can be used to analyze requirements elicitation interviews and to understand the narratives and "stories" that emerge during those interviews.

Accelerated Requirements Method (ARM)

The ARM process [Hubbard 2000] is a facilitated requirements elicitation and description activity. It includes three phases:

1. Preparation phase
2. Facilitated session phase
3. Deliverable closure phase

The ARM process is similar to JAD but has certain significant differences from the baseline JAD method, which contribute to its uniqueness. For example, in this process, the facilitators are content neutral, the group dynamic techniques used are different from those used in JAD, the brainstorming techniques used are different, and the requirements are recorded and organized using different conceptual models.

3.5.2 Elicitation Evaluation Criteria

Following are example evaluation criteria that may be useful in selecting an elicitation method, although you could certainly use other criteria. The main point is to select a set of criteria and to have a common understanding of what they mean.

- *Adaptability.* The method can be used to generate requirements in multiple environments. For example, the elicitation method works equally well with a software product that is near completion as it does with a project in the planning stages.
- *Computer-aided software engineering (CASE) tool.* The method includes a CASE tool.
- *Stakeholder acceptance.* The stakeholders are likely to agree to the elicitation method in analyzing their requirements. For example, the method isn't too invasive in a business environment.
- *Easy implementation.* The elicitation method isn't overly complex and can be properly executed easily.

- *Graphical output.* The method produces readily understandable visual artifacts.
- *Quick implementation.* The requirements engineers and stakeholders can fully execute the elicitation method in a reasonable length of time.
- *Shallow learning curve.* The requirements engineers and stakeholders can fully comprehend the elicitation method within a reasonable length of time.
- *High maturity.* The elicitation method has experienced considerable exposure and analysis with the requirements engineering community.
- *Scalability.* The method can be used to elicit the requirements of projects of different sizes, from enterprise-level systems to small-scale applications.

Note that this approach presumes that all criteria are equally important. If some criteria are more important than others, a weighted average can be used. For example, availability of a CASE tool might be more important than graphical output. A typical weighting scheme could consider criteria to be "essential" with weight 3, "desirable" with weight 2, and "optional" with weight 1. The elicitation methods can then be ranked using a tabular form, as shown in Table 3–4. The example in Table 3–4 is not intended to be an actual recommendation to use a specific method. You can develop your own comparison criteria and ratings.

In our case studies, we decided to use JAD, ARM, and IBIS on three different projects. These three methods were subjectively ranked to be the most suitable candidates for the case studies, given the time and effort constraints for the project. We considered not just the total score: The learning curve was an important factor, and the team attempted to select methods that were not too similar to one another, so as to have some variety. In our case studies, we had the most success using ARM to identify security requirements. Detailed results for all three methods can be found in the Requirements Engineering section of the Build Security In Web site [BSI 10].

Additional Considerations

It is possible that a combination of methods may work best. You should consider this option as part of the evaluation process, assuming that you have sufficient time and resources to assess how methods

Table 3–4: *Comparison of Elicitation Methods*

	Misuse Cases	*SSM*	*QFD*	*CORE*	*IBIS*	*JAD*	*FODA*	*CDA*	*ARM*
Adaptability	3[a]	1	3	2	2	3	2	1	2
CASE tool	1	2	1	1	3	2	1	1	1
Stakeholder acceptance	2	2	2	2	3	2	1	3	3
Easy implementation	2	2	1	2	3	2	1	1	2
Graphical output	2	2	1	1	2	1	2	2	3
Quick implementation	2	2	1	1	2	1	2	2	3
Shallow learning curve	3	1	2	1	3	2	1	1	1
High maturity	2	3	3	3	2	3	2	2	1
Scalability	1	3	3	3	2	3	2	1	2
Total Score	**18**	**18**	**17**	**16**	**22**	**19**	**14**	**14**	**18**

a. 3 = Very good; 2 = Fair; 1 = Poor.

may be combined and to actually combine them. You should also consider the time necessary to implement a particular elicitation method and the time needed to learn a new tool that supports a method. Selecting a requirements elicitation method that meets the needs of a diverse group of stakeholders aids in addressing a broader range of security requirements.

3.6 Requirements Prioritization

Once you have identified a set of security requirements, you will usually want to prioritize them. Given the existence of time and budget constraints, it can be difficult to implement all requirements that have been elicited for a system. Also, security requirements are often implemented in stages, and prioritization can help to determine which ones should be implemented first. Many organizations pick the lowest-cost requirements to implement first, without regard to importance. Others pick the requirements that are easiest to implement—for example, by purchasing a COTS solution. These ad hoc approaches are not likely to achieve the security goals of the organization or the project.

To prioritize security requirements in a more logical fashion, we recommend a systematic prioritization approach. This section discusses a tradeoff analysis that you can perform to select a suitable requirements prioritization method and briefly describes a number of methods. We also discuss a method of prioritizing requirements using AHP. More extensive coverage of this material is available elsewhere [Chung 2006].

While results may vary for your organization, the discussion of the various techniques should be of interest. Much work needs to be done before security requirements prioritization is considered a mature area, but it is one that we must start to address.

3.6.1 Identify Candidate Prioritization Methods

A number of prioritization methods have been found to be useful in traditional requirements engineering and could potentially be used for developing security requirements. We briefly mention here the binary search tree, numeral assignment technique, planning game, the 100-point method, Theory-W, requirements triage, Wiegers' method, requirements

prioritization framework, and AHP. Further information can be found on the Build Security In Web site and in the references.

Binary Search Tree (BST)

A binary search tree is an algorithm that is typically used in a search for information and can easily be scaled to be used in prioritizing many requirements [Ahl 2005]. The basic approach for requirements is as follows, quoting from [Ahl 2005]:

1. Put all requirements in one pile.
2. Take one requirement and put it as the root node.
3. Take another requirement and compare it to the root node.
4. If the requirement is less important than the root node, compare it to the left child node. If the requirement is more important than the root node, compare it to the right child node. If the node does not have any appropriate child nodes, insert the new requirement as the new child node to the right or left, depending on whether the requirement is more or less important.
5. Repeat Steps 3 and 4 until all requirements have been compared and inserted into the BST.
6. For presentation purposes, traverse through the entire BST in order and put the requirements in a list, with the least important requirement at the end of the list and the most important requirement at the start of the list.

Numeral Assignment Technique

The numeral assignment technique provides a scale for each requirement. Brackett proposed dividing the requirements into three groups: mandatory, desirable, and unessential [Brackett 1990]. Participants assign each requirement a number on a scale of 1 to 5 to indicate its importance [Karlsson 1995]. The final ranking is the average of all participants' rankings for each requirement.

Planning Game

The planning game is a feature of extreme programming [Beck 2004] and is used with customers to prioritize features based on stories. It is a variation of the numeral assignment technique, where the customer distributes the requirements into three groups: "those without which

the system will not function," "those that are less essential but provide significant business value," and "those that would be nice to have."

100-Point Method

The 100-point method [Leffingwell 2003] is basically a voting scheme of the type that is used in brainstorming exercises. Each stakeholder is given 100 points that he or she can use for voting in favor of the most important requirements. The 100 points can be distributed in any way that the stakeholder desires. For example, if there are four requirements that the stakeholder views as having equal priority, he or she can put 25 points on each. If there is one requirement that the stakeholder views as having overarching importance, he or she can put 100 points on that requirement. However, this type of scheme works only for an initial vote. If a second vote is taken, people are likely to redistribute their votes in an effort to move their favorites up in the priority scheme.

Theory-W

Theory-W (also known as "win-win") was initially developed at the University of Southern California in 1989 [Boehm 1989; Park 1999]. This method supports negotiation to solve disagreements about requirements, so that each stakeholder has a "win." It relies on two principles:

1. Plan the flight and fly the plan.
2. Identify and manage your risks.

The first principle seeks to build well-structured plans that meet predefined standards for easy development, classification, and query. "Fly the plan" ensures that the progress follows the original plan. The second principle, "Identify and manage your risks," involves risk assessment and risk handling. It is used to guard the stakeholders' "win-win" conditions from infringement. In win-win negotiations, each user should rank the requirements privately before negotiations start. In the individual ranking process, the user considers whether he or she is willing to give up on certain requirements, so that individual winning and losing conditions are fully understood.

Requirements Triage

Requirements triage [Davis 2003] is a multistep process that includes establishing relative priorities for requirements, estimating the resources

needed to satisfy each requirement, and selecting a subset of requirements to optimize the probability of the product's success in the intended market. This technique is clearly aimed at developers of software products in the commercial marketplace. Davis's more recent book [Davis 2005a] expands on the synergy between software development and marketing; we recommend that you read it if you are considering this approach. Requirements triage is a unique approach that is worth reviewing, although it clearly goes beyond traditional requirements prioritization to consider business factors as well.

Wiegers' Method

Wiegers' method relates directly to the value of each requirement to a customer [Wiegers 2003]. The priority is calculated by dividing the value of a requirement by the sum of the costs and technical risks associated with its implementation [Wiegers 2003]. The value of a requirement is viewed as depending on both the value provided by the client to the customer and the penalty that occurs if the requirement is missing. Given this perspective, developers should evaluate the cost of the requirement and its implementation risks as well as the penalty incurred if the requirement is missing. Attributes are evaluated on a scale of 1 to 9.

Requirements Prioritization Framework

The requirements prioritization framework and its associated tool [Moisiadis 2000, 2001] includes both elicitation and prioritization activities. This framework is intended to address the following issues:

- Elicitation of stakeholders' business goals for the project
- Rating the stakeholders using stakeholder profile models
- Allowing the stakeholders to rate the importance of the requirements and the business goals using a fuzzy graphic rating scale
- Rating the requirements based on objective measures
- Finding the dependencies between the requirements and clustering requirements so as to prioritize them more effectively
- Using risk analysis techniques to detect cliques among the stakeholders, deviations among the stakeholders for the subjective ratings, and the association between the stakeholders' inputs and the final ratings

AHP

AHP is a method for decision making in situations where multiple objectives are present [Saaty 1980; Karlsson 1996, 1997]. This method uses a "pair-wise" comparison matrix to calculate the value and costs of individual security requirements relative to one another. By using AHP, the requirements engineer can confirm the consistency of the result. AHP can prevent subjective judgment errors and increase the likelihood that the results are reliable. It is supported by a stand-alone tool as well as by a computational aid within the SQUARE tool.

3.6.2 Prioritization Technique Comparison

We recommend comparing several candidate prioritization techniques to aid in selecting a suitable technique. Some example evaluation criteria are provided here:

- *Clear-cut steps:* There is clear definition between stages or steps within the prioritization method.
- *Quantitative measurement:* The prioritization method's numerical output clearly displays the client's priorities for all requirements.
- *High maturity:* The method has had considerable exposure and analysis by the requirements engineering community.
- *Low labor-intensity:* A reasonable number of hours are needed to properly execute the prioritization method.
- *Shallow learning curve:* The requirements engineers and stakeholders can fully comprehend the method within a reasonable length of time.

Note that this simple approach does not consider the importance of each criterion. It is also possible to construct a weighted average when comparing techniques. For example, maturity may be of greater importance than learning curve. This difference could be taken into account by weighting the results and ranking the various criteria as "essential" with weight 3, "desirable" with weight 2, and "optional" with weight 1. A comparison matrix used in a case study is shown in Table 3–5. This example is not intended to be an actual recommendation to use a specific technique; you can develop your own comparison criteria and ratings.

For one of our case studies, we considered the numeral assignment technique (NAT), Theory-W (TW), and AHP. The results of the comparison are summarized in Table 3–5.

Table 3–5: *Comparison of Prioritization Techniques for a Case Study*

Selection Criteria	*NAT*	*TW*	*AHP*
Clear-cut steps	3[a]	2	3
Quantitative measurement	3	1	3
Maturity	1	3	3
Labor-intensive	2	1	2
Learning curve	3	1	2
Total Score	**12**	**8**	**13**

a. 3 = Very good; 2 = Fair; 1 = Poor.

We decided to use AHP as a prioritizing method. This decision was made on the basis of the results shown in Table 3-5 comparison, recognizing that the rankings are subjective. Factoring into the rationale behind choosing AHP were the team members' familiarity with the method, its quantitative outputs, and its structure in providing definite steps for implementation. The detailed case study results are described in [BSI 11].

3.6.3 Recommendations for Requirements Prioritization

Prioritization of security requirements is an important activity. We recommend that stakeholders select candidate prioritization techniques, develop selection criteria to pick one, and apply that methodology to decide which security requirements to implement when. During the prioritization process, stakeholders can verify that everyone has the same understanding about the security requirements and further examine any ambiguous requirements. After everyone reaches consensus, the results of the prioritization exercise will be more reliable.

3.7 Summary

In this chapter, we initially focused on the role of misuse and abuse cases, which can motivate the identification and documentation of security requirements. The examination of SQUARE focused on the process to support security requirements engineering. We also explored a method for selecting a requirements elicitation process and provided experimental results for several candidate elicitation processes. We then focused on methods for prioritizing requirements and described the results of a case study in this area.

On the Build Security In Web site, we also discuss the Comprehensive, Lightweight Application Security Process (CLASP) approach to security requirements engineering, core security requirements artifacts, and the use of attack trees in security requirements engineering [BSI 12]. Formal specification approaches to security requirements, such as REVEAL and Software Cost Reduction (SCR), are also useful in this regard. The higher levels of the Common Criteria provide similar results [BSI 13]. Another article on BSI discusses the use of integer programming for optimizing investment in implementation of security requirements [BSI 14].

Although security requirements engineering is an area of active research, many promising techniques have already emerged that you can use to identify the requirements needed to improve your software products. It seems obvious that systematic and thorough identification of security requirements in the early stages of the SDLC will result in more secure software and reduced security costs later on.

Here are our recommendations for software project managers who wish to pay more attention to security requirements engineering:

- Review your existing development practices. Do you have development practices that are specific to requirements engineering? If not, put a standard requirements engineering practice in place.
- If you have an existing requirements engineering process, does it address security requirements? If not, use the material presented in this chapter and elsewhere to decide which steps need to be taken to produce good security requirements. If your process does address security requirements, have you considered the end-user's and attacker's perspectives, in addition to the perspectives of other stakeholders?

- There is no one-size-fits-all answer to security requirements engineering. You need to analyze your projects to figure out which ones need more attention to security requirements (such as mission-critical systems) and which ones are less critical (such as administrative systems that don't contain sensitive data). Note, however, that even unimportant systems can potentially provide an attacker with an indirect means to access more critical systems and sensitive data.
- Start small. Try out your new and improved practices on a couple of systems under development so that you can debug the process without making all of your projects part of a grand experiment. The case study results described on the BSI Web site give guidance on how to go about this [BSI 10, 11].
- Document your results—both positive and negative—and use them to improve your processes. As you know, development processes need to be revisited periodically, just like everything else we do in software and security engineering.
- Don't be a victim of NIH ("not invented here"). If someone else has an approach to security requirements engineering that could potentially be useful to you, give it a try. It's not necessary to reinvent the wheel every time.