10

A Lightweight Approach to Secure Software Engineering

M. G. Jaatun, J. Jensen, P. H. Meland and I. A. Tøndel
SINTEF ICT

CONTENTS

10.1 Introduction

Secure software engineering[1] is much more than developing critical software. History has shown us that software bugs and design flaws also represent exploitable security vulnerabilities in seemingly innocuous applications such as web browsers and PDF document viewers. This implies that there is a need for a well-balanced amount of security awareness in all software development projects right from the beginning.

Most software developers are not primarily interested in (or knowledgeable about) security; for decades, the focus has been on implementing as much functionality as possible before the deadline, and then patch whatever bugs there may be when it's time for the next release or hotfix. However, it is slowly beginning to dawn on the software engineering community that security is important also for software whose primary function is not related to security.

There are clear indications that significant cost savings and other advantages are achieved when security analysis and secure engineering practices are introduced early in the development cycle, and that the number of serious security defects can be significantly reduced with a minimum of extra costs. However, having a clear security focus is not easy, and today there are very few people that master both the art of software and security engineering. There is thus a need for closer collaboration and knowledge transfer between the two factions.

SODA is an approach to inject secure software engineering practices into existing software development processes. The SODA target group is the "ordinary" developer, who is not primarily interested in (or knowledgeable about) security, but must focus on designing/implementing as much functionality as possible before the deadline is passed and/or the budget is exhausted.

SODA is based on the following assumptions:

1. A developer will not try to learn or memorize security knowledge prior to starting the development.
2. There should be no significant change in the way developers work.
3. There must be good tool support that enhances security during development, preferably integrated into the current development tools.

In the following sections, we will present how this philosophy is reflected in approaches for asset identification, security requirements elicitation, security design, and security testing. As can be seen from the grayed-out parts of Figure 10.1, we will not cover secure coding, deployment or monitoring, but these topics are covered by several easily accessible books (e.g., [5, 6]).

[1]This chapter is primarily based on results from the SODA project, as documented in a series of papers [1, 2, 3, 4]. For more detailed references and background, please refer to these original papers.

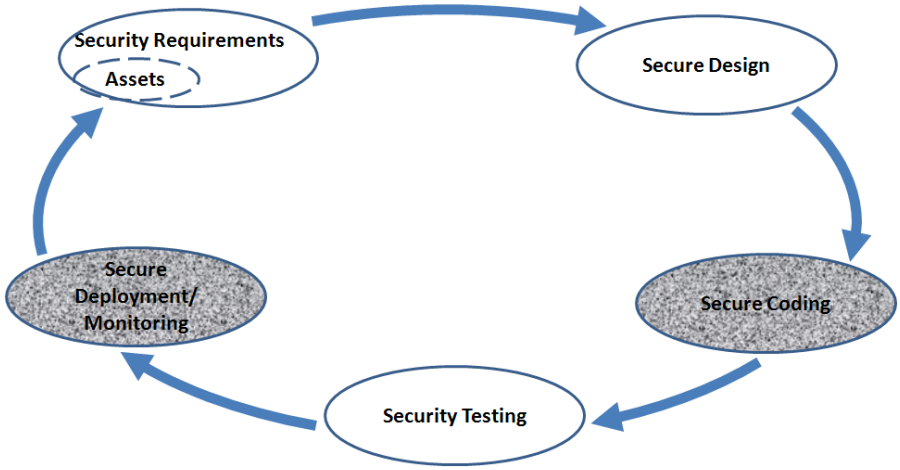

FIGURE 10.1
The main phases of the SODA approach to secure software engineering.

10.2 Assets

10.2.1 Asset Identification

The concept of "assets" is central to the very idea of security – we need security because we have something that needs protection. This "something" is what we collectively refer to as our assets. Thus, asset identification is a crucial component of the requirements phase – specifically, security requirements are primarily needed in order to specify what we need to do to protect our assets, and this will obviously be impossible to do properly unless we know what these assets are. To highlight the importance of the asset identification phase, we detail it here first, separately from the main security requirements phase, which is described in Section 10.3.

The goal of the method we describe in the following is to discover all the assets that are relevant for the system being developed, and facilitate a prioritization process in order to identify which assets have a higher (or lower) priority with respect to security. Strictly speaking, our primary concern is to identify the assets that are *most important* – if we overlook assets that *don't* need protection, we can still sleep at night.

Before asset identification takes place, the main security objectives of the software to be developed should be identified. By security objectives, we mean high-level security requirements or goals identified by customers, and any security requirements coming from standards, policies, or legislation. The results of asset identification should be used as a basis for identifying threats, where

attack trees or similar are created based on the most important assets identified. Security requirements are then elicited based on threat analysis.

10.2.2 Asset Identification in Practice

The SODA asset identification method helps to establish an overview of the assets of a system and their different requirements for protection. This information makes it easer to prioritize security requirements (either informally, or as part of a formal risk analysis process) later on. We suggest to look at the value of the assets both from the user's and the system owner's point of view, but also from the view of an attacker. To identify assets, one can use functional requirements of the system in combination with brainstorming techniques. For each asset, one should then make a judgment regarding the different stakeholders' priority of the confidentiality, integrity, and availability of this asset. When assigning priorities, one should use predefined categories, for instance, high, medium, and low.

When an asset identification has been performed, you should have an overview of which properties of what assets are most important to protect. This information should be used to prioritize requirements and to identify where to focus the effort when it comes to identifying more detailed requirements.

10.2.2.1 Key Contributors

The asset identification process should be performed by a diverse group composed of

- The developer's requirements team
- Other developers (not part of the core requirements team)
- Security experts (if available – security expertise will be a bonus in this process, but our method is designed to also work without it)
- Customers and/or end-users (if applicable)

Generally, anyone that could contribute with ideas on assets may contribute, but the total group should not exceed eight persons (plus facilitator). We assume that the participants either are familiar with the system before the asset identification starts, or that a short briefing on the main system characteristics is given as part of the introduction.

This method can be used in any project. For large projects, one may however need to use more coarse assets than in smaller projects, or execute the method several times on a subset of the application domain.

10.2.2.2 Step 1: Brainstorming

Our brainstorming technique may be considered an amalgamation of traditional brainstorming and "brainwriting." The purpose of brainstorming is gen-

erally to generate ideas on a given topic; in our case, the topic is restricted to answering the question "what are our assets." By using this technique, one is able to involve different types of persons in a creative idea-generating process, without putting too many restrictions on the end result.

As preparation for the brainstorming session, sticky notes[2] and felt-tip pens/markers must be made available for all participants, and somewhere to put the sticky notes during brainstorming must be provided (e.g., a large sheet of paper to put on the wall).

1. Present the process and the rules to follow while brainstorming:
   - •Everybody shall participate
   - •No discussion/criticism during the brainstorming
   - •One should build on ideas presented by others
2. Give out sticky notes and markers to all participants.
3. Decide on a time limit for individual brainstorming (e.g., 5 minutes).
4. Formulate a question on which to brainstorm (e.g., "What are our assets?"), write it down and place it somewhere visible to everybody in the group.
5. All participants write down their ideas – one idea in large letters per note (the note should be readable from a distance of several meters).
6. When time is up, everybody presents their ideas to the group by placing their notes on, for instance, a piece of paper on the wall. Often it is advantageous to do this in a structured way: Everybody takes turns presenting their ideas. One is only allowed to present a limited number of ideas at the time, and the remaining ideas must wait until the next round.
7. Group the ideas and eliminate duplicates. Everybody should participate in this step.
8. Document the result, for example, with a camera.

Note that although Wilson [7] warns that a trained facilitator is necessary to ensure success of a brainstorming session, we have found that even with just "normal" participants (and appointing one facilitator), there is tangible value to be had from brainstorming. Furthermore, this is also to a great extent *learning by doing* – if a development organization frequently employs brainstorming in relevant situations, the individual participants will in time become reasonably confident (if not to say skilled) as facilitators.

[2] *For example, Post-it Notes*®, a registered trademark of 3M (http://www.3m.com). We have found that generic-brand sticky notes also work. (We are also aware that there are various computer-based brainstorming tools that may be used instead of our paper-based method, but this is simply a matter of preference – the general process remains the same.)

TABLE 10.1
Asset prioritization table

ASSETS	**STAKEHOLDERS' PRIORITY**		
	Focus: protection. What is most important to protect from stakeholders' point of view?		**Focus: attacks.** What is most interesting/valuable for an attacker?
Description	**System user**	**System owner**	**Attacker**
<asset 1>	<C-? I-? A-?>	<C-? I-? A-?>	<C-? I-? A-?>
<asset 2>	<C-? I-? A-?>	<C-? I-? A-?>	<C-? I-? A-?>
...	...	...	...

In a small group where the participants know one another well, it may be more cost-effective with respect to time to let each participant express their ideas orally (one at a time), and assign one participant (or facilitator) to write the ideas on a whiteboard as they are presented. In this case, duplicates are avoided, and it may be easier (quicker) to get new ideas regarding assets based on the assets that are being presented.

10.2.2.3 Step 2: Assets from Existing Documentation

Once the brainstorming session is finished, it is a good idea to examine any available functional requirements or functional descriptions of the system to determine whether any important assets have been overlooked. In some cases, this may inspire a second round of brainstorming.

10.2.2.4 Step 3: Categorization and Prioritization

Once a list of assets is available, the assets must be categorized and prioritized with respect to security. This should be performed by the same group that participated in the asset identification, possibly augmented by management participation. (Management should possibly *not* be invited to the brainstorming session itself, since one of the pitfalls identified by Wilson [7] is inviting "anyone [...] who is feared by the other members.")

We recommend assigning priorities from three stakeholders' perspectives:

- The System User
- The System Owner
- The Attacker

As can be seen from Table 10.1, the different stakeholders' priority of the assets is described with three letters:

C: Confidentiality

I: Integrity

A: Availability

These letters can then get assigned a value indicating the importance of confidentiality, integrity, or availability for this asset. We maintain that the traditional CIA triad is enough for this purpose – additional properties like nonrepudiation will represent special cases that should be treated separately.

1. Decide which categories to use to represent priorities. In most cases it will be adequate with three (qualitative) levels:

   H:High

   M:Medium

   L:Low

2. For each asset, make a judgment regarding the different stakeholders' priority of the confidentiality, integrity, and availability of this asset. If, for example, confidentiality is not an issue for an asset, it is not necessary to include this property and assign a level to it (the same goes for integrity and availability).
3. After values for the importance of CIA have been assigned for all stakeholders of all assets, calculate a ranking sum for each security property of each asset by adding 3 for every High, 2 for every Medium, and 1 for every Low. For instance, if the asset "Web Server" is listed as A-M, C-M I-H A-H, C-M I-H A-H, the web server gets a Confidentiality rank of $0+2+2 = 4$, and so forth.
4. Create three ordered lists with the assets prioritized according to Confidentiality, Integrity, and Availability.
5. If many assets have been identified, consider pruning the assets with consistently low priorities.

If management has not been involved up to this point, they should be given the opportunity to comment on the asset tables. For small systems where a limited number of assets are identified, the final prioritization may be performed manually.

The success of the method is very much dependent on the individuals participating in asset identification, as the types of assets identified will depend on their competence and main focus. Since the method is based on brainstorming, which is not a structured method, it may be beneficial to "tune" the process by utilizing checklists or predefined questions in the brainstorming activity.

Using functional requirements as a starting point comes with the risk of not covering abstract assets such as the company's reputation, the safety of employees, and availability and connectivity of resources. We believe, however, that for this lightweight approach, most of the assets of this type can be

indirectly covered by looking at the different actors' value of the assets. When stating the value of an asset from the owner's point of view, reputation should be part of the evaluation.

In the SODA approach, information on an asset is limited to values representing the importance of the confidentiality, integrity, and availability of this asset. This is done to keep the method lightweight, and it is what is needed for prioritization. The reasons behind these values and the criteria used are however lost. This may reduce the possibility to reuse the results later, and makes it harder to compare results of different sessions since the criteria may be different.

An advantage of including the attacker's perspective is that this indirectly covers some assets that are otherwise easily overlooked (or difficult to relate to). A specific example of this is the "Reputation" asset: In the past, it may not have mattered to a company if someone uses their file servers without permission for storing data, as long as they behave themselves and do not create problems for legitimate users. (The example is somewhat construed, since I cannot imagine a system administrator ever "not caring" about illegitimate users on her system; however, this effectively would have been the reaction of the local police force should the incident have been reported: "Sooo... nothing was *actually* stolen ...?") However, this changes dramatically if the company risks being exposed in the tabloids as a haven for file-sharers and other deviants – suddenly protecting the "File Server" asset becomes much more important, implicitly because of the "Reputation" asset.

10.2.3 Example

We will now illustrate how the asset identification technique works through development of an imaginary example, which we will call the LyeFish tool.

The idea behind LyeFish is to provide a tool for amateur chefs who want to experiment with the effect of lye solutions on ichthyoids. It is basically a publicly accessible web resource with open content, but where users can log on to receive more personalized service. LyeFish shall assist the users in selecting techniques by offering a set of questions about the user's preferences and create a profile based on the answers.

LyeFish shall be an independent and self-contained application – meaning that it does not depend on any other systems to be useful to its users. However, LyeFish may contain recommendations of other products that are useful for applying the individual techniques.

The security objectives (ref. Section 10.3.2) that provided the context for the asset identification were

- Integrity of the application
- Hosting organization's IT regulations and security policy

A brainstorming session as described in Section 10.2.2.2 produces the assets

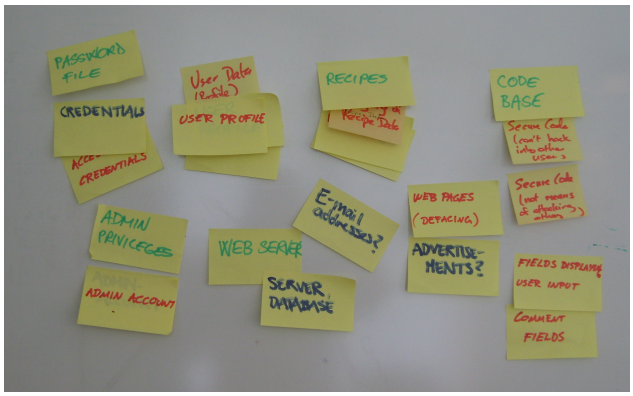

FIGURE 10.2
Result of the brainstorming session.

TABLE 10.2
Asset prioritization table

ASSETS	**STAKEHOLDERS' PRIORITY**		
	Focus: protection. What is most important to protect from stakeholders' point of view?		**Focus: attacks.** What is most interesting/valuable for an attacker?
Description	**System user**	**System owner**	**Attacker**
Code base		C-L I-M A-M	C-M I-H A-L
Data	I-M A-M	I-H A-M	I-L A-L
Profile	C-M I-H A-L	C-M I-H A-L	C-L I-L A-L
Credentials	C-H I-H A-L	C-H I-H A-L	C-H I-H A-L
Admin. account	C-H I-H A-H	C-H I-H A-M	C-M I-H A-L
Web Server	A-M	C-M I-H A-H	C-M I-H A-H

depicted in Figure 10.2. We then proceed immediately to classification and prioritization.

The identified assets and the results of the prioritization process are shown in Table 10.2. Reputation was also identified as an asset, but it proved difficult to fit this into the mold, and it was therefore left out of the table. However, it was agreed that damage to the integrity of the application would also damage our reputation (as system owner).

As can be seen, availability of the personalized service is not considered of high importance, since the open content will still be available. Confidentiality is also not considered relevant for the open content.

Finally, prioritization is performed for each of the security categories Confidentiality, Integrity, and Availability by assigning each H the value 3, each

TABLE 10.3
Calculated asset ranking

Confidentiality		Integrity		Availability	
ASSETS	**Sum**	**ASSETS**	**Sum**	**ASSETS**	**Sum**
Admin. account	9	Admin. account	9	Web Server	8
Credentials	9	Credentials	9	Admin. account	6
Profile	5	Profile	7	Data	5
Web Server	4	Web Server	6	Code base	5
Code base	3	Data	6	Credentials	3
Data	0	Code base	5	Profile	3

M the value 2 and each L the value 1. A missing category for any stakeholder counts as 0. This produces three prioritized lists of assets; on per security category (C,I,A), where the highest sum gives the highest priority.

In this example, note that the confidentiality and integrity of the administration account is considered equivalent (in terms of ranking score) to the other credentials in the system, but as security professionals, we intuitively give the administration account slightly higher priority. Note also that when it comes to availability, the web server itself comes out on top – this is reasonable, since the web application may deliver considerable benefit even if the administrator is unable to access it.

10.3 Security Requirements

10.3.1 Description

Information security requirements are important in all software engineering projects, not only to ensure the correct level of security in the end product, but also to avoid implementing security solutions that turn out to be a bad fit. Since security thus is important also 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 both easy to understand, and easy to use! Although formal methods undoubtedly have their merits, their use is precluded in this context.

Before we dive into the description of the security requirements process, we will briefly describe some artefacts which are typically used and/or created when eliciting security requirements.

Misuse cases (see Figure 10.3) [8] extend the regular use case diagrams with negative use cases (misuse cases) that specify behavior not wanted in the

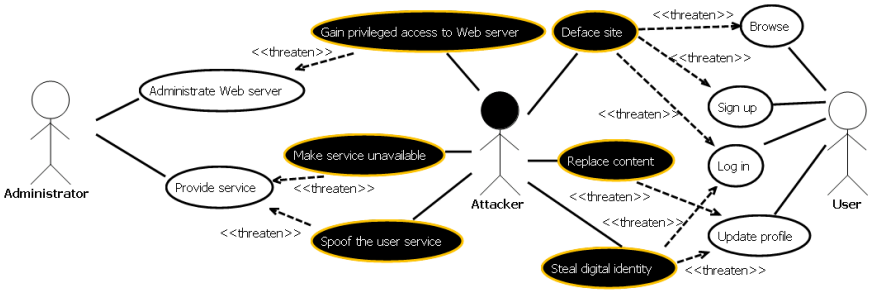

FIGURE 10.3
Misuse case diagram for a publicly available web application.

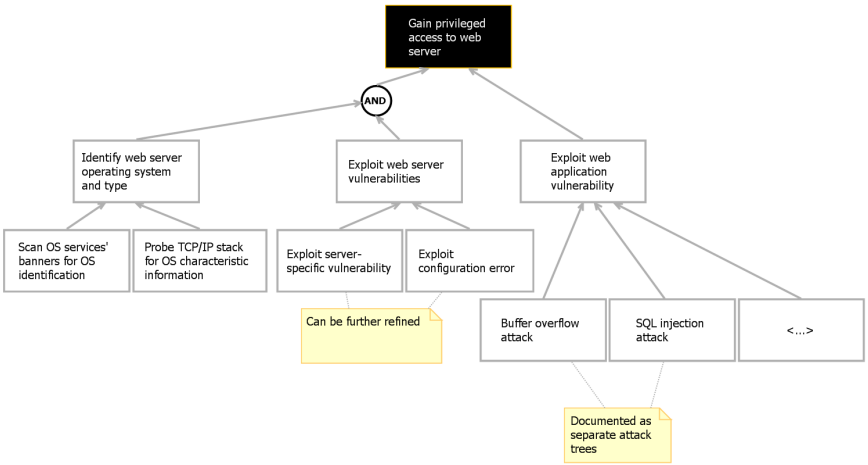

FIGURE 10.4
Attack tree detailing an attack on a web server.

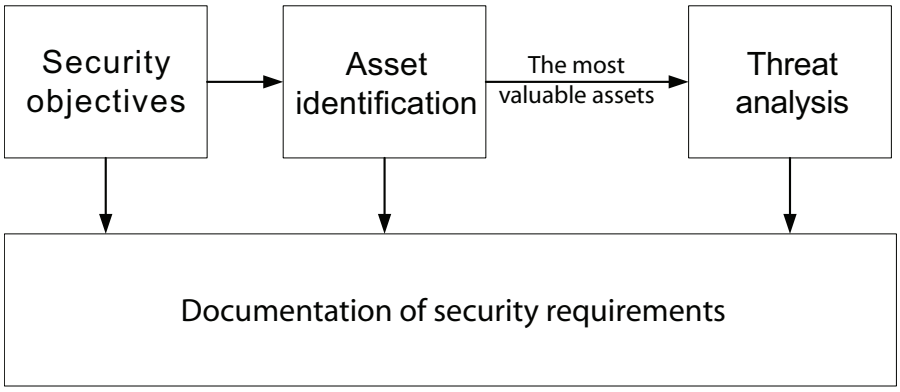

FIGURE 10.5
Core requirements phase.

system. Use cases can *mitigate* misuse cases, meaning that a use case can be a countermeasure against a misuse case – thereby reducing the chances that the misuse case succeeds. Misuse cases can *threaten* a use case, meaning that the use case is exploited or hindered by a misuse case.

Abuser stories were first introduced by Peeters [9] as an agile counterpart to misuse cases. An abuser story is a brief and informal description of how an attacker may abuse the system at hand. Abuser stories are not yet widely used, but there are some experience reports [10, 11] that show that variants have been used successfully in agile software development projects.

Attack trees Attack trees [12] represent attacks/threats against a system in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes (see Figure 10.4). The diagrams can be used both in the requirements and design phases. Trees can be represented graphically or can be written in outline form. It is also possible to add information on, for example, cost of attack.

The main focus of the SODA security requirements phase is identification of security objectives, assets, and threats; this results in the steps for identification of security requirements that are illustrated in Figure 10.5.

10.3.2 Security Objectives

The aim of this step is to identify the paramount security requirements; that is, the requirements that are most important to customers, and the requirements that must be met to comply with relevant legislation, policies, and standards. This is necessary to set boundaries and constraints in order to prioritize security efforts and make necessary trade-offs later. Identifying security objectives

consists of two main activities: Identification of the customer's need for security, and identification of relevant legislation, policies, standards, and best practices that apply to the system/module. An important part of the process of eliciting security requirements is customer meetings that focus on security issues, and we will provide concrete tips when it comes to how to prepare for such a meeting. We also give examples of where to look for requirements from legislation, policies, and standards.

10.3.3 Asset Identification

When an asset identification has been performed (see Section 10.2), you should know which assets are most important to protect, and hopefully also which *properties* of these assets are the most important. This information should be used to prioritize requirements and to identify where to focus the effort when it comes to identifying more detailed requirements.

10.3.4 Threat Analysis and Modeling

Swiderski and Snyder [13] list the following purposes of threat modeling:

- Understand the threat profile of a system.
- Provide recommendations/solutions for secure design and implementation.
- Discover potential vulnerabilities.
- Provide feedback for the application security life cycle.

The aim here is to identify the main threats to the system/module. We recommend basing the identification of threats on the most important assets identified and the STRIDE categories (spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege). The most important threats identified should then be further elaborated using attack trees. The attack trees identified at this stage should not be making too many assumptions regarding design decisions that have not been made yet. The attack trees will therefore be less detailed than what may be needed in later stages of the development process and may therefore need to be further elaborated in later phases. While it is certainly possible to draw threat trees and misuse case diagrams by hand, special-purpose tools such as SeaMonster (see Section 10.4.4.3) often makes the job easier, particularly when the diagrams need to be updated.

If the developers have access to a vulnerability repository (see Section 10.5.4), it should be consulted here as part of the threat analysis, and also revisited in the design phase.

10.3.5 Documentation of Security Requirements

The main aims of this activity are to make the security requirements visible, show which security requirements are high priority, and arrange for traceability and follow-up of requirements. We suggest to describe all requirements in one place, either in a separate document or as part of a general requirements document, to be able to keep an overview of all requirements.

A good security requirement is similar to pornography, in that it is difficult to define, but we recognize it when we see it.[3] We recommend to describe requirements that are focused on *what* should be achieved – not *how*. Also, negative requirements ("It should no be possible to ...") should be avoided, since they are not testable. The following is an example of a good security requirement: "Only hashed passwords shall be stored in the user database." We also recommend to give each requirement an identifier, specify the source of the requirement, and state the requirement's priority. For each requirement, it should also be possible to add information on how the requirement is followed up during development (requirement tracking).

10.3.6 Variants Based on Specific Software Methodologies

The core requirements phase is, as much as possible, not tied to a specific software development methodology. This results in the recommendations being usable for a broad group of software developers, but some developers could benefit from using other techniques that fit better with their current methodology. The technique where this is most obvious is *threat analysis*, where we currently recommend attack trees. If the developer has already used UML use cases to describe functional requirements, misuse cases will probably be a better choice than attack trees. For developers using, for example, eXtreme Programming, abuser stories will probably be the better choice. Agile developers will also probably prefer a less rigid way of documenting security requirements – for instance, by using the abuser stories for this directly. For developers using these methodologies, the recommended techniques will therefore differ from what is specified as the *core requirements phase*.

10.3.7 LyeFish Example Continued

Having identified the most important assets in Section 10.2.3, we now look into possible threats to these assets before going on to specify security requirements. We do this with the help of misuse case diagrams and attack trees.

We start out by sketching some high-level properties of the LyeFish tool, normally as a coarse use case diagram (remember to avoid making assumptions about design decisions that have not yet been made!). We then perform

[3] Paraphrased from Potter Stewart's concurrence in the *Jacobellis vs Ohio* case (1964).

another brainstorming exercise, this time focusing on how an attacker might abuse our system, and extend our use case diagram into a misuse case diagram.

In Figure 10.3, we have presented (parts of) a misuse case diagram for the LyeFish tool. From the diagram, we can see that the actor "Administrator" administrates the web server, and generally "provides service" through the LyeFish application. The actor "User" is someone who uses the LyeFish application via a web browser, and among the many things a User might do, the diagram illustrates browsing recipes, signing up for a personalized account, logging into an existing account, and updating personal profile. On the right side of the diagram can be seen an "Attacker" actor who might be interested in various nefarious activities, including gaining privileged access to the web server, making the service unavailable, and stealing the identity of innocent Users.

In Section 10.2.3, we determined that the administrator account and the web server itself were the most important assets, and we could argue that getting privileged access to the web server is a good step on the way to compromising the administrator account. We have therefore detailed an attack tree for this in Figure 10.4. Note that neither the misuse case diagram nor the attack tree presented can be considered complete, but are illustrations which can be used as starting points. Note also that it pays to keep misuse case diagrams and attack trees small; if they get too large, it is easy to get lost. Partly for this reason, we have decomposed the attack "Exploit Web Application Vulnerability" into specific attacks "Buffer Overflow," "SQL injection," and so forth, and each of these are documented (elsewhere) as a separate attack tree.

On the left side of Figure 10.4, there are two branches connected by an "and" symbol, meaning both branches must be accomplished in order to have a successful attack. If we can minimize the possibility of exploiting a web server vulnerability, that goes a long way, and thus a reasonable requirement would be: "A regime for timely application of security updates shall be implemented for the web server." If possible, it would be even better if "timely" could be quantified better.

Since the right branch covers web application vulnerabilities, this calls for a bunch of "classical" software security requirements, for example, "All input from web users must be validated on the server side."

10.4 Secure Software Design

The SODA approach is also applicable to software development projects where architecture and design are central.

10.4.1 Security Architecture

The architecture describes a system on an abstract level, while leaving the implementation details unspecified. The traditional security architecture deals with system level security mechanisms and issues, such as security perimeters, cryptography, access control and authorization. Having this separation can be unfortunate, as the security should be embedded into the overall software and system architecture, creating a secure architecture. This should be done by showing how the architecture, and its components satisfy both the functional and nonfunctional security requirements. The software architecture should include countermeasures to compensate for vulnerabilities or inadequate assurances in individual components or cross-component interfaces, for example, by isolating components.

The architecture and the more detailed design will usually consist of a set of (hierarchical) diagrams and documents that describe the structure and behavior of software and its environment. There are several notations and techniques for creating these, with the UML flavors as the industry leader. Several security specific extensions with tool support for UML have been created.

Creating a secure architecture and design is the overall activity in this life cycle phase and relies on and iterates with requirements specification and implementation. The next sections describe techniques that are to be used as a part of architecture and design.

10.4.2 Security Design Guidelines

Security design guidelines should be considered as a broad category of theoretical information that comes in handy when creating secure applications. These typically span from less formal best practices, principles, and rules-of-thumb to different kinds of policies, rules, regulations, and standards. Howard and Lipner [14] say that secure design best practices focus on "good security hygiene" within the application. However, the challenge is to know what good hygiene is before you start doing the "dirty work." Forcing too much theoretical information about ways to incorporate security is not very efficient. To be aligned with the SODA assumptions, we have chosen to focus on two specific kinds of guidelines and best practices; namely, security design principles and security patterns.

10.4.2.1 Security Design Principles

Security design principles are a specific type of guidelines and practices. They are proven rules for improving the security posture of an application, and in order to be useful, the principles must be applied to specific problems. This is the great advantage with them since they can be identified during the requirements phase doing threat modeling. There exist a large number of such principles, and even though just reading through them once in a while will

improve security consciousness, the real value is added when they are directly used to identify weaknesses and argue for architecture and implementation decisions.

The security design principles in Table 10.4 are built on the idea of simplicity and restriction [15].

10.4.2.2 Security Patterns

A security pattern is a well-understood solution to a recurring security problem, and encourages effective reuse for building in robustness. Software design patterns have become widely accepted after the Gang of Four published their very influential book [16] on this topic in the middle of the nineties, and there exists a vast number of patterns for software development; see, for example, Hillside[4] for an extensive online library. However, while some security patterns take the form of traditional design patterns, not all of them are design patterns.

Security patterns are usually divided into different types and categories, typically:

- Structural, behavioral, and creational security patterns encompass design patterns, such as those used by the Gang of Four. They include diagrams on relationships between entities and descriptions of interaction and object creation.
- Available System patterns is a subtype of structural patterns and they facilitate construction of systems which provide predictable uninterrupted access to the services and resources they offer to users.
- Protected System patterns is another sub-type of structural patterns that facilitate construction of systems which protect valuable resources against unauthorized use, disclosure, or modification.
- Antipatterns are ways of not doing things based on things that have failed in the past or invalid assumptions. An antipattern should also include a solution, for example, reference to a working pattern.
- A mini-pattern is a shorter, less formal discussion of security expertise in terms of just a problem and its solution. Programming language-specific patterns are also known as idioms.
- Procedural patterns are patterns that can be used to improve the process for development of security-critical software. They often impact the organization or management of a development project and are therefore not security design patterns.

Table 10.5 gives examples of common security patterns, their type, and a short description.

[4] http://www.hillside.net/patterns/.

TABLE 10.4
Examples of design principles

Principle	Definition
Principle of Least Privilege	The *principle of least privilege* states that a subject should be given only those privileges that it needs.
Principle of Fail-Safe Defaults	The *principle of fail-safe defaults* states that, unless a subject is given explicit access to an object, it should be denied access.
Principle of Economy of Mechanism	The *principle of economy of mechanism* states that security mechanisms should be as simple as possible.
Principle of Complete Mediation	The *principle of complete mediation* requires that all accesses to objects be checked to ensure that they are allowed.
Principle of Open Design	The *principle of open design* states that the security of a mechanism should not depend on the secrecy of its design.
Principle of Separation of Privilege	The *principle of separation of privilege* states that a system should not grant permission based on a single condition.
Principle of Least Common Mechanism	The *principle of least common mechanism* states that mechanisms used to access resources should not be shared.
Principle of Psychological Acceptability	The *principle of psychological acceptability* states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present.

TABLE 10.5
Security pattern examples

Name(s)	Type	Abstract
Single access point, Login window, One way in, Guard door, Validation screen	Protected system pattern	*Providing a security module and a way to log into the system. Set up only one way to get into the system, and if necessary, create a mechanism for deciding which subapplications to launch.*
Account lockout, Disabled password	Behavioral pattern	*Account lockout protects customer accounts from automated password-guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed.*
Standby, disaster recovery backup site	Available system pattern	*Structures a system so that the service provided by one component can be resumed from a different component.*
Maginot line	Antipattern	*To use a security solution that worked in the past, but is now outdated.*
Share responsibility for security, Nonseparation of duty	Procedural pattern	*This pattern makes all developers building an application responsible for the security of the system.*

10.4.3 Threat Modeling and Security Design Review

Threat modeling is an iterative process, continuously revisited throughout the software life cycle. We introduced threat modeling/analysis in Section 10.3 as an important part of the requirements phase, and we return to it here.

At this point, we know more about the system we are building than at the beginning of the requirements phase, and we use this knowledge to refine our threat models, identifying what functionality and which assets an attacker can take advantage of. The software design should be evaluated from an attacker's point of view. This process will result in a threat model document that can be used by developers to identify which threats are present, and which steps should be taken to mitigate the associated risks.

An architecture and design review helps you validate the security-related design features of your application before you start the development phase. This allows you to identify and fix potential vulnerabilities before they can be exploited and before the fix requires a substantial re-engineering effort.

Security design review is a technique that can be used to discover vulnerabilities that have been overlooked earlier in the design phase of the project. Dowd et al. [17] suggest identifying the trust boundaries in the design, and identifies six main elements to review for each boundary; authentication, authorization, accountability, confidentiality, integrity, and availability. These ideas are similar to using checklists during the security design review. Table 10.6 shows a simplified version of a checklist focused on Web application security [18].

10.4.4 Putting It into Practice – More LyeFish

We now assume that we have performed asset identification and security requirements elicitation for the LyeFish tool and proceed with secure design.

10.4.4.1 Applying Security Design Principles

Memorizing all design principles you come over is of little use. Principles are a type of knowledge that can only be fully understood through experience. In order to gain such knowledge, we recommend the following approach:

- Start with a few principles at a time. Do not try to comprehend them all at a time. You can, for instance, try to pick out the three most important ones for your current project.
- Try to understand the reason for the principles. It can slow down development if you apply principles just for the sake of it, without understanding why. It can even lead to whole layers of your application that serve no real purpose. Once you understand the reasoning behind the principles, it becomes much easier to choose how and where to apply principles.
- The most important thing is that you try - the rest comes with experience.

TABLE 10.6
Checklist for security review

Input validation	All entry points and trust boundaries are identified by the design.
	Input validation is applied whenever input is received from outside the current trust boundary.
	The design addresses potential SQL injection issues.
	The design addresses potential cross-site scripting issues.
	The design does not rely on client-side validation.
Authentication	The design partitions the Web site into public and restricted areas.
	The design identifies the mechanisms to protect the credentials over the wire (SSL, encryption, and so on).
	Account management policies are taken into consideration by the design.
	The design ensures that minimum error information is returned in the event of authentication failure.
	The identity that is used to authenticate with the database is identified by the design.
	The design adopts a policy of using least-privileged accounts.
Authorization	The role design offers sufficient separation of privileges (the design considers authorization granularity).
	The design identifies code access security requirements. Privileged resources and privileged operations are identified.
	All identities that are used by the application are identified and the resources accessed by each identity are known.
Sensitive data	The design identifies the methodology to store secrets securely.
	The design identifies protection mechanisms for sensitive data that are sent over the network.
	Secrets are not stored unless necessary.
Cryptography	The methodology to secure the encryption keys is identified.
	Platform-level cryptography is used and it has no custom implementations.
	The design identifies the key recycle policy for the application.
Exceptions	The design outlines a standardized approach to structured exception handling across the application.
	The design identifies generic error messages that are returned to the client.
Auditing & logging	The design identifies the level of auditing and logging necessary for the application and identifies the key parameters to be logged and audited.
	The design identifies the storage, security, and analysis of the application log files.

> You will make mistakes, but your programs should still be superior to the ones you developed before you started thinking about principles.

For the LyeFish tool, it is natural to start with the first principle in Table 10.4, the *principle of least privilege*. We can apply this on several levels, from making sure that the LyeFish application can run as an unprivileged process, to not granting various users access to more information than they need. The *principle of fail-safe defaults* also applies, since we don't want, for example, errors in the application to give external users direct access to privileged web server content.

10.4.4.2 Making Use of Security Design Patterns

As already mentioned, several existing security patterns can be found in books, articles, and on the Web. The challenge is making these more readily available for developers, primarily from within their development tools. We see procedural patterns more as guidelines, so for the design phase, we focus on design patterns for security. We have defined two practical uses for these, namely:

- Using a security design pattern as a starting template when creating new design documents.
- Applying security design patterns to your existing design documents.

Several of the leading CASE tools support the instantiation of diagrams based on design patterns. This functionality allows you to create design stubs, making it faster and easier to make use of the proven good solutions. However, security design patterns are unfortunately not found among the default patterns for most of these tools.

For LyeFish, the two first entries in Table 10.5 are clearly relevant, in that we have the opportunity of logging into the system, and want to be able to lock out accounts that are targeted for brute-force password guessing.

10.4.4.3 Make Use of Tools for Threat Modeling

SODA threat modeling during design builds on the initial threat modeling from the requirements phase. This basis should be used as input to help select specific security design principles, guidelines, and patterns. To aid this process, it can be useful to use tools such as SeaMonster,[5] which supports several types of threat modeling notations/artifacts. Throughout the design phase, threat modeling should be continuously revisited in order to identify new threats or to mitigate the existing ones. This way, the threat model itself can be used as documentation for the security activities and countermeasures applied during development life cycle. SeaMonster facilitates reuse and sharing of threat models, and supports linking to external resources for information about threats and attacks (compliant with SODA assumption 2).

[5] http://sourceforge.net/projects/seamonster/.

10.4.4.4 Performing Security Review

A security and architecture review should be performed by someone else other than the designers of the target system. If there exists a dedicated security team or someone with that assigned role, that would be the obvious choice, but such a review could also be performed by regular designers and developers (with a little help). The main point is to have unbiased eyes look at and question the design artifacts that have been produced so far.

If there exist architecture and design documentation, a good start would be to go through these and see which security features are reflected here. The next step would then be to go through your findings with one of the designers to check whether your results match the intention.

If the documentation is sparse, outdated or nonexisting, you should separately interview two or more designers about the security features and compare their responses. If there are many differences, put the designers together and go through the mismatches one by one.

You do not have to wait until the end of a phase to perform a security review. A review can (and should) be performed early and several times in order to detect design flaws as soon as possible.

10.5 Testing for Software Security

Traditional software testing is mainly an exercise in Quality Assurance; "Does the application meet all the [functional] requirements [...] ?"[19]. If the requirements say "To get B, input A," the tester will input A, and if B is output, the test is categorized a success. Software security testing is more about testing things that *shouldn't* happen, however, and since the variations of possible incorrect input are practically infinite, you can never test all permutations. The security testing phase of SODA thus focuses on penetration testing of applications or parts of applications before deployment.

10.5.1 Background

According to McGraw [6], it is necessary to involve two different security testing approaches: Functional and adversarial (see Table 10.7). As mentioned, functional security testing of security mechanisms is not a controversial strategy, but many run-of-the-mill applications will have very few (or none) of these mechanisms, rendering the *functional* security testing a relatively manageable task. Since our focus is security testing of these "average" applications, we are thus primarily interested in the *adversarial* approach. Techniques for software security testing are thoroughly covered by several books [19, 20, 21, 22]. Several tools have also been developed to help testers, for example, tools that

TABLE 10.7
Approaches to security testing [6]

No.	Approach	Why
1	Functional security testing	To determine whether security mechanisms such as access control and cryptography settings are implemented and configured according to the requirements.
2	Adversarial security testing	To determine whether the software contains vulnerabilities by simulating an attacker's approach – based on risk-based security testing.

focus on error handling, monitoring of environmental interaction, and tools for intercepting and modifying traffic between server and client.

Software security testing tools and techniques have to constantly evolve to be able to detect vulnerabilities that can be utilized in new types of attacks. But there are even bigger challenges:

- Security testing is often neglected in development projects.
- When penetration testing is performed, software development organizations tend to treat the results as complete bug reports – when each item on the list has been crossed off, the system is considered "secure."
- There is seldom any feedback loop to facilitate learning from the penetration testing process to the development organization, which leads to the same type of software vulnerabilities being introduced in later versions of the software.

Security testing is a typical last-minute activity, and the resources available are scarce. Risk-based testing, which means focusing the testing effort on critical functions, therefore becomes important. Some types of security vulnerabilities are more serious and/or more common than others, and statistics and rankings like OWASP Top 10 and the SANS Top 25 can be used to focus testing. Some applications, or parts of applications, can also be more likely to cause problems:

- The highest risk is experienced by web facing systems, large code size applications, and new applications.
- Complexity may be an indicator for future security problems, using tools to measure cyclomatic complexity.
- Error handling routines should be an important focus in testing since many

security failures occur in stressed environments. This is often neglected during testing because it is difficult to simulate such conditions.

In general, we advocate threat modeling as a basis for risk-based testing, focusing on application entry points. Traditional black box testing usually takes an outside-in approach where the testers do not have previous knowledge of the software to be tested. Risk-based security testing, on the other hand, implies that the test process is driven by some form of risk-related input, where the risk evaluation is based on previous knowledge of the software. The risk management process gives an indication of where an attack on the newly developed software is most likely to succeed, thus testing can be focused on the most vulnerable code.

Evidence for the benefits of risk-based testing is provided by Potter and McGraw [23]. In a case study, they did both functional and risk-based testing on smart card technologies. Via the functional tests, they found that security mechanisms often were satisfactorily implemented with respect to the defined requirements. However, when the units that previously passed the functional test were exposed to a risk-based security testing approach, all of them failed! These findings emphasize the importance of structuring and prioritizing the security tests based on risk. However, Potter and McGraw also express that risk-based security testing relies on expertise and experience, something that may be a problem in small organizations since most regular developers are not primarily interested in security issues.

10.5.2 The Software Security Testing Cycle

The software security testing cycle differs from the traditional testing cycle primarily in one aspect: Security testing does not have a clear-cut fulfillment criterion, and is therefore open-ended – we can go on testing for weeks, and still not be done. Furthermore, there are frequently limited resources available for testing, and this results in a need to prioritize testing efforts. Thus, the first step in the cycle as illustrated in Figure 10.6 is to define the focus and scope of the impending security testing activity.

We acknowledge the importance of taking a risk-based approach to security in all software development phases, including software security testing. This is even more important in a lightweight approach, where there is no way all security aspects can be fully addressed and tested. Our focus is on giving concrete guidelines as to how risk-based security testing can be achieved in ordinary software engineering projects. In Section 10.5.3, we describe how concrete results from security techniques applied in the requirements, design, and implementation phases can be used as a basis for deciding what to focus on in testing activities. These risk management activities that are tied to the specific application developed should be used together with known risk factors such as complex components, web-facing components, error handling, and so forth. In addition, it should be taken into account where we typically have failed in the past.

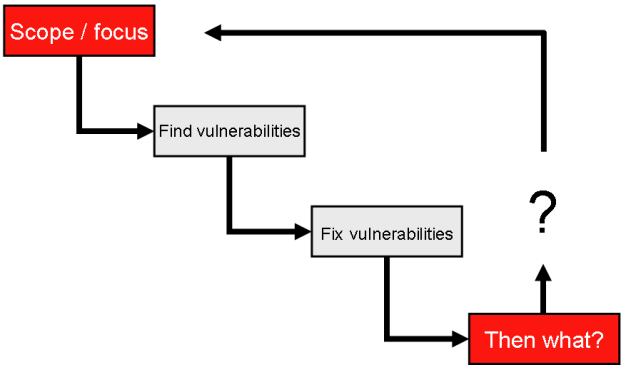

FIGURE 10.6
Software security testing cycle.

As described in Section 10.5.1, various testing techniques and tools are readily available, and we therefore do not focus on the second and third step of the testing cycle, that is, finding and fixing bugs. Instead, we choose to stress that it does not stop here! For each vulnerability that is discovered by testing, special care should be taken to verify whether similar vulnerabilities exist in other parts of the same application, and root causes identified. The results from security testing must be used to improve software development, in all phases, to make sure the same mistakes are not repeated in the next version or the next project.

10.5.3 Risk-Based Security Testing

Artifacts relevant to a risk-based testing will be produced in all phases of a software development life cycle. The lightweight techniques assisting the requirements and design phases are no different, and they output artifacts such as lists of assets, security requirements, and threat models.

The following sections describe how each of these help targeting the security testing to maximize the return on the testing effort.

Assets and Security Requirements Getting access to (or otherwise attacking) the important assets identified as part of the security requirements elicitation should be a main focus in testing activities. The security requirements will mainly give input to the functional security testing activities, but will also point to high-risk areas that should be considered for adversarial security testing.

Exploiting Threat Models in Security Testing Threat models will provide the most important input to the adversarial security testing, and help to visualize important parts of an application's attack surface; thus pointing

to the areas in the code that are most exposed to attacks. Testing must be performed on these areas to ensure that the most likely attacks are not achievable.

Static Code Analysis Results Ideally, vulnerabilities reported by the static analysis tools are fixed immediately. However, it is common knowledge that these tools are not able to discover every possible vulnerability. Code segments where the returned number of vulnerabilities are above average may indicate either that the programmer has done a poor job, or that it was a particularly difficult segment to code. In both cases, one should expect more vulnerabilities to lie dormant, and thus these parts should be thoroughly tested. Also note that found vulnerabilities ("positives") sometimes are erroneously classified as "false positives."

Choice of Language Programming languages have different inherent security properties, and thus choice of language will influence the testing [24]. Since, for example, C/C++ is vulnerable to buffer overflows, this is something you would want to test for; however, this would not be relevant if, for example, Java, Pascal, or Ada were the programming language of choice.

It is also possible to take this idea a step further and look at which development environments that are used. Content Management Systems (CMS), for instance, are popular tools for web application development, and several of these have been reported to contain vulnerabilities. By searching through vulnerability databases (see below) for your development environment, a list of already known vulnerabilities can be found. These vulnerabilities should be tested to determine whether your application is at risk.

Dynamic Code Analysis and Fuzzing There are also various dynamic code analysis tools on the market, which can be useful if the development team has access to them. In particular, fuzzing tools [25] can be a golden opportunity to narrow the "infinite permutation gap" mentioned earlier (i.e., that testers can never manually test all possible combinations of input to a given program). Fuzzing is a modern variant of what used to be called "the kindergarten test"[6] in some circles. The fuzzer application will input random data via various interfaces in an automated fashion, and will record any unexpected behavior and/or failures.

10.5.4 Managing Vulnerabilities in SODA

Information about publicly known vulnerabilities are available at several sources like the Security Focus vulnerability database and the associated mailing list *Bugtraq*, the National Vulnerability Database hosted by NIST and sponsored by the Department of Homeland Security/US-CERT, and the

[6] Type random gibberish at the prompt, and see what happens.

Open Source Vulnerability Database. Common Vulnerabilities and Exposures (CVE), hosted by the MITRE Corporation, provides common identifiers for vulnerabilities and exposures. For a more thorough treatment of public vulnerability repositories and mailing lists, see Ardi et al.[26].

Knowledge about publicly known vulnerabilities is important and can be utilized by software development organizations. Different organizations have different characteristics, however; culture, knowledge level, or special properties of the applications being developed can be the reason why some vulnerabilities are more common than others. Knowledge of what are the most commonly introduced vulnerabilities can be used to focus testing, but also to improve the software development process.

By analyzing vulnerabilities, it is possible to understand what the organization is doing wrong, and compensate through the use of secure development techniques. To better enable learning from our own mistakes, we suggest organizing information about all vulnerabilities found in a vulnerability repository. This repository should include vulnerabilities found during security testing, but also design flaws found during design review, coding mistakes found by static analysis tools or code reviews (if applied), and vulnerabilities found after deployment. It is important to learn from vulnerabilities regardless of when they are detected.

The goals for the vulnerability repository include:

- **Improve the ability to produce secure software:** By using the vulnerability repository actively to guide the security development process in the organization, it should be possible to reduce the number of vulnerabilities in software, especially vulnerabilities that have traditionally been common or have been given focus because of the high risk involved.

- **More cost-effective handling of vulnerabilities:** Focus on common vulnerabilities should result in these vulnerabilities being avoided altogether or detected at earlier stages where the cost of fixing vulnerabilities is at a minimum.

- **Measure progress:** The vulnerability repository can be used to measure progress, that is, whether vulnerabilities are detected sooner in the development process, or whether the number of vulnerabilities are reduced. Such measurements can be a motivation factor. Note however that too much focus on reduced number of vulnerabilities can result in less effort when it comes to finding vulnerabilities, and thereby reduced quality of testing. Finding many vulnerabilities is a good thing and should also be appreciated.

Information from the vulnerability repository should be utilized at all stages of the development process in order to avoid or detect the vulnerabilities as early as possible:

- **Requirements engineering:** Vulnerabilities that are common and potentially high risk can be used as input to general software development policies

that apply to all applications developed by the organization. Example: All applications shall have proper input handling. These general policies will then be used as a basis for security requirements together with customer requirements, asset identification, and threat modeling activities [2].

- **Design:** Knowledge of common vulnerabilities can guide software designers to make more secure design choices, like choosing appropriate security design guidelines, principles, and patterns. As an example: Statistical evidence that an organization's software is susceptible to SQL injection attacks may be used as an argument to include the Intercepting Validator security pattern defined by Steel et al. [27]. A design review should also profitably be based on past experiences.

- **Implementation:** The choice of language and the use of frameworks can influence which types of implementation vulnerabilities that are common. If large groups of vulnerabilities can be removed by changing, for example, programming language this should be taken into account when making such decisions. Knowledge of common vulnerabilities can also be used to focus code reviews, if the organization is performing such reviews. Finally, knowledge of common implementation errors can be used to tune the rule-sets used in the static code analysis tools.

- **Testing:** Information on where we typically have failed in the past should be used as input when prioritizing testing efforts. Concrete vulnerabilities as well as statistics on which vulnerability categories are most common should be utilized. This type of information can be obtained from the suggested vulnerability repository.

If lack of knowledge and training is identified as a cause for important groups of vulnerabilities, this information should be used to focus training initiatives. Concrete vulnerabilities should then be utilized for motivation.

To limit the work related to registering and follow-up of vulnerabilities, we suggest to strive toward only registering the information you intend to use. We suggest the following information as a minimum:

- Date – to be able to measure progress over time.

- Vulnerability category – to be able to know what areas to focus on in improvement activities.

- Where – to be able to know what portions of the code and what aspects of the application to focus on when improving security development techniques.

- In which phase and with which technique it was detected – to make sure the vulnerability is tested for in the future and to be able to see whether the same type of issue is detected earlier in future projects.

- Root cause – to use as input to improve the security development process.

- Risk – to prioritize what to focus on, but also to see if the risk posed by the vulnerabilities detected decreases over time.
- Countermeasure – to learn how this type of vulnerability can be prevented or avoided, for example, by referring to a relevant security pattern.

In addition, it may be of interest to record who introduced the vulnerability to allow each developer to learn directly from their own mistakes. However, it is important to consider possible side effects, such as employees feeling they are being publicly embarrassed and become uncomfortable with their working environment.

For the registration of vulnerabilities, we suggest to utilize predefined categories to ease aggregation and searchability of information, and use free text for details. Regarding vulnerability category, it is advantageous to use existing vulnerability taxonomies like the 7+1 kingdoms defined by Tsipenyuk et al. [28], and detail these if necessary. For describing risk it is possible to utilize the Common Vulnerability Scoring System [29]. By representing the vulnerabilities in a standard way, it will be easier to share vulnerability information in an anonymized and generalized form, so that they can be integrated in a public or federated repository.

10.5.5 Example – Testing LyeFish

We will now apply the risk-based testing approach to the LyeFish tool. In Table 10.3, we found that the administrator account and the web server itself were the highest-ranked assets of LyeFish. The administrator login interface will therefore be a logical starting point for adversarial testing.

The attack tree in Figure 10.4 next practically gives us a step-by-step procedure for attacking the web server. There are also automated tools that can be wielded against particular web servers. We can also point a fuzzer at the LyeFish web interface, and leave it running for a given time, recording all identified problems.

Finally, identified vulnerabilities (i.e., tests that compromise security) are recorded in the vulnerability repository.

10.6 Summary

With an increasing number of threats to software, security must be considered from the very beginning of every software development project. However, most software security tools and methodologies have been created with traditional security-critical projects in mind. With security becoming a concern in average projects, these methods are not always appropriate, especially for developers without the proper security background. Our motivation has been to

establish a lightweight approach that should be used in every project, without consuming more resources than necessary.

This chapter has presented a lightweight approach to identifying assets, eliciting security requirements, performing secure software design, and finally security testing. Secure coding could have been a chapter of its own, but the interested reader is encouraged to explore this further in the references [5, 6]).

10.7 Further Reading and Web Sites

The SHIELDS project worked on detecting known security vulnerabilities from within design and development tools, and the project results are archived at `http://shields-project.eu`. This web site also contains many other useful links.

There are a number of databases of software vulnerabilities available on the internet; we consider the most important to be the *National Vulnerability Database* (`http://nvd.nist.gov/`), the *Open Source Vulnerability Database* (`http://osvdb.org/`), and the *Common Vulnerabilities and Exposures* (`http://cve.mitre.org`). In addition, there is a lot of useful, although less structured, information available in the *Security Focus vulnerability Database* at `http://www.securityfocus.com/archive/1`.

The Open Web Application Security Project maintains the "OWASP Top 10" list of the ten most critical web application security risks at `http://www.owasp.org/index.php/OWASP\Top_Ten_Project`. This list is augmented by the "CWE/SANS Top 25‘" most dangerous software errors at `http://www.sans.org/top25-software-errors/`. Combined, the OWASP Top 10 and the CWE/SANS Top 25 represent a minimum baseline that all software engineering projects should be aware of in order to avoid embarrassing security errors.

Bibliography

[1] Martin Gilje Jaatun and Inger Anne Tøndel. Covering your assets in software engineering. In *The Third International Conference on Availability, Reliability and Security (ARES 2008)*, pages 1172–1179, Barcelona, Spain, 2008.

[2] Inger Anne Tøndel, Martin Gilje Jaatun, and Per Håkon Meland. Security Requirements for the Rest of Us: A Survey. *IEEE Software*, 25(1), 2008.

[3] Per Håkon Meland and Jostein Jensen. Secure software design in practice. In *Availability, Reliability and Security, (ARES 2008). Third International Conference on*, pages 1164–1171, Barcelona, Spain, March 2008.

[4] Inger Anne Tøndel, Martin Gilje Jaatun, and Jostein Jensen. Learning from software security testing. In *Software Testing Verification and Validation Workshop, 2008, ICSTW'08*, pages 286–294, April 2008.

[5] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2003.

[6] Gary McGraw. *Software Security–Building Security In*. Addison-Wesley, 2006.

[7] Chauncey E. Wilson. Brainstorming pitfalls and best practices. *interactions*, 13(5):50–63, 2006.

[8] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[9] Johan Peeters. Agile Security Requirements Engineering. In *Proceedings of The 2005 Symposium on Requirements Engineering for Information Security (SREIS)*, 2005.

[10] Gustav Boström, Jaana Wäyrynen, Marine Bodén, Konstantin Beznosov, and Philippe Kruchten. Extending XP practices to support security requirements engineering. In *SESS '06: Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, pages 11–18, New York, NY, USA, 2006. ACM Press.

[11] Vidar Kongsli. Towards agile security in web applications. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 805–808, New York, NY, USA, 2006. ACM.

[12] Bruce Schneier. Attack Trees–Modeling security threats. *Dr. Dobb's Journal*, July 2001.

[13] Frank Swidersky and Window Snyder. *Threat Modeling*. Microsoft Professional, 2004.

[14] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[15] M. A. Bishop. *Computer Security: Art and Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[17] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment*. Addison-Wesley, 2007.

[18] J. D. Meier. Web application security engineering. *IEEE Security and Privacy*, 4(4):16 – 24, 2006.

[19] Chris Wysopal, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. *The Art of Software Security Testing: Identifying Software Security Flaws*. Symantec Press, 2006.

[20] James A. Whittaker and Herbert H. Thompson. *How to Break Software Security*. Addison-Wesley, 2003.

[21] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[22] Tom Gallagher, Lawrence Landauer, and Bryan Jeffries. *Hunting Security Bugs*. Microsoft Press, 2006.

[23] Bruce Potter and Gary McGraw. Software security testing. *Security & Privacy Magazine, IEEE*, 2(5):81–85, Sept.-Oct. 2004.

[24] K. M. Goertzel, T. Winograd, H. L. McKinley, L. Oh, M. Colon, T. McGibbon, E. Fedchak, and R. Vienneau. Software Security Assurance. Technical report, Information Assurance Technology Analysis Center and Data (IATAC) and Analysis Center for Software (DACS), 2007.

[25] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[26] Shanai Ardi, David Byers, Per Håkon Meland, Inger Anne Tøndel, and Nahid Shahmehri. How can the developer benefit from security modeling? In *The Second International Conference on Availability, Reliability and Security (ARES 2008)*, Vienna, Austria, 2007.

[27] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall, 2005.

[28] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *Security & Privacy Magazine, IEEE*, 3(6):81–84, Nov.-Dec. 2005.

[29] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *Security & Privacy Magazine, IEEE*, 4(6):85–89, Nov.–Dec. 2006.