

Part 1

Introduction

In this first part, we set the stage for this book. We present how we think about security, development, and how they fit together. We analyze where problems tend to occur and what we think can be done about it. The opening chapter covers these aspects, together with an example of what we mean by secure by design.

We finish this part with an intermission chapter that is more of a light read. Here, we introduce some of the ideas of the next part through a case study from a client we've worked with. So, let's get started with how security and development fit together, and the basic ideas behind secure by design.

1

Why design matters for security

This chapter covers

- Viewing security as concerns, not features
- Design and why it's important for security
- Building in lots of security by focusing on good design
- Addressing the Billion Laughs attack

Imagine yourself setting up a typical software project. You assemble a team of developers, testers, and domain experts and start outlining the key requirements. With input from stakeholders, you come up with a list of important attributes: performance, security, maintainability, and usability. As with many software projects, quality takes top priority, time to market is of the essence, and you need to stay within budget. You decide to be proactive and add security features to your backlog, and some of the other team members come up with a good list of security libraries you can use in your code. After the initial planning, you get the project up and running and start implementing features and business functionality. The team is motivated and delivers features at a good pace.

Although you know you should *think security all the time*, it gets in the way of other tasks you're focused on. In addition to that, most of the time you aren't working on internet-facing code anyway, so those web security libraries you thought about using

don't really fit. Plus, the security-related tasks in the backlog keep getting lower priority compared to the business functionality. After all, time is tight, and it doesn't matter if the system is secure if the features the users need aren't there. Business functionality is where the money is, and no user is going to thank you for putting CSRF tokens in your login form.¹ Besides, you can always go back and deal with lower priority tasks later.

As a developer, you feel the responsibility of security is a burden you'd rather not have on your shoulders. You think it'd be better if the company brought in a security expert on a permanent basis as part of the development team. Developers are experts at crafting good code, building scalable architectures, and using continuous delivery, not waving magic wands to cast spells that can defend against evil hackers in black hoodies. You have never understood why security has to be so secretive in the first place, and it's much more fulfilling to create than it is to destroy. The project must move forward, so you keep your focus on the top of the backlog and implementing features.

After some time, your software is ready to go into production. Your project's future can now play out in a couple of different ways. One way is that you conduct a security audit and a *penetration test*.² But the security review report finds vulnerabilities that are considered to be severe enough that you must address them before deploying to production. This sets your project back a couple of weeks, or maybe even months, with lost revenue as a consequence. If you're unlucky, solving the issues involves rewriting the entire program from scratch, so the stakeholders decide to scrap the project, and it never makes it into production.

Another scenario is that a security review is never conducted, and you deploy into production. Users start to use your service, and all is well, until one day you find your service has made it into the news after being hacked and having all its user data leaked. Those hard-earned users are now abandoning your service quicker than rats leaving a sinking ship.

Although this is a fictional story, it's not that far from reality. During our careers, we've seen similar scenarios play out more than once. A couple of interesting things are at play here, and some questions arise:

- Why is it that security tasks always get lower priority compared to other tasks?
- Why are developers in general so seemingly uninterested in security?
- Experts keep telling developers to think more about security, so why isn't everyone doing it?
- Why don't managers realize they need to include security experts in the team just as they put testers in the team?

Literature and experts have been telling us to focus more on security for a long time. Alas, we keep seeing news about systems being hacked every so often. Something is clearly not working.

IMPORTANT In order to efficiently and effortlessly create secure software, you need to have a mindset different from what you might be used to.

¹ For more about CSRF tokens, see https://en.wikipedia.org/wiki/Cross-site_request_forgery.

² A penetration test is a test performed on a system to uncover possible security weaknesses.

What if there were a different way to approach software security that allowed you to avoid many of the problems we see in our industry today? We believe that in order to efficiently and effortlessly create secure software, you need to have a mindset that might be different from what you're used to—a mindset where you focus more on design rather than on security.

This might sound counterintuitive at first, but in this chapter, we'll explain what we mean by the word *design* and why it's important for security. We'll discuss some of the shortcomings of the traditional approach to software security and show you how you can use design to overcome those issues. We'll also provide a couple of examples of how to apply these ideas in the real world in order to give you a first taste of some of the concepts covered in the upcoming chapters.

1.1 Security is a concern, not a feature

A productive way to view security is as a concern—as in, “we’re concerned about security.” But it’s not uncommon to come across situations where security is described as a set of features. The difference is that even when security features address a specific security problem, your concern about security may not have been met. To illustrate how security is a concern rather than a feature, let’s start with a historical example. Let’s go back in time to one of the first recorded bank robberies in history to see how security features like high-quality locks don’t matter if hinges are weak. In the example, the features implemented didn’t prevent the robbery, so the concern for security wasn’t met.

1.1.1 The robbery of Öst-Götha Bank, 1854

It is the night of March 25, 1854, and the Swedish Öst-Götha Bank is soon to be robbed. A military corporal and former farmer, Nils Strid, walks silently up to the bank with his companion, the blacksmith, Lars Ekström. The outer door to the bank office is locked, but the key hangs outside on a nail if you know where to look. The bank has also invested in high-quality locks for the vault—more or less impossible to pick. But for blacksmith Lars, it’s not a big job to splinter the hinges and open the vault door backward. The two perpetrators walk away with the entire treasury of the bank: 900,000 riksdaler, the official Swedish currency at the time.³

For years, this was one of the largest heists in history. Not until the great train robbery in Buckinghamshire, England, at Bridego Railway Bridge in 1963, would the loot be of a similar size. In Sweden, the burglars left behind a single three-riksdaler banknote,⁴ together with a message with a silly rhyming verse:

Vi länsat haver Östgötha Bank och mången rik knös torde blivit pank. Vi lämna dock en tredaler kvar ty hundar pissar på den som inget har.

We now have plundered Östgötha Bank, and many moneybags will become broke. However, we leave a three-daler behind, because dogs piss on those who have naught.

³ Comparing this to the value of money nowadays is hard, but a comparable sum would be in the range of 5 to 10 million dollars.

⁴ Yes, there actually were notes with the denomination of three.

Apart from being an interesting historical event, the robbery is also interesting from a security point of view in two different ways: one legal and one technical. From a legal perspective, the robbery resulted in new laws mandating a certain level of bank security. These laws forced the banks at that time to adhere to some level of security awareness and practices. The first, passed in the following year, 1855, was one of the earliest examples of regulatory security. From a technical perspective, the robbers consistently attacked the bank's weak spots: the office door was locked, but the key was poorly hidden; the vault locks were of high quality, but the hinges could be broken.

What this story spotlights is how security can be viewed as a set of features—locks and hinges. In our example, using high-quality locks gave the perception of security, but security wasn't implemented by that feature as such. Having high-quality locks isn't sufficient if the key hangs on a nail or if the vault door hinges are weak. Rather than treating security as a set of features, it's more fruitful to understand it as a concern that should be met.

Had the bank viewed security as a concern, it would have asked, "How do we stop people from walking away with the bank's money?" The answer wouldn't have been with a lock; it should have included keeping the office key elsewhere or checking if there were other ways to force the vault door. The bank's owners might have come up with novel ideas about alarms. They might have invented some of the theft-detering mechanisms that emerged during the coming century, but they wouldn't have relied on just having a lock on the door.

Now let's return from nineteenth-century banking to the contemporary world of software development. It's time to see how the idea of security as a feature or a concern applies to your projects today. In the next section, we'll show you how you can turn from specifying security as a feature to identifying security as a concern.

1.1.2 **Security features and concerns**

Software is often described in the language of features (or what you can do with the product). For example, this is an app that lets you share a shopping list; this is a site where you can upload photos for others to see and comment on; this is a program for creating presentation slideware. Software is also described in this way in formal contexts.

Many methodologies have their primary focus on what the system should do—the functional side. The Rational Unified Process (RUP), which still influences a lot of software development, puts the major focus on the functionality in the form of *use cases*. Other considerations like response time or capacity required are put in a peripheral section called supplementary specifications. In the agile community, the dominating format for describing what's to be done in the next sprint (or comparable) is a *user story* in a format along the lines of "As a such-and-such user, I'd like this feature so that I reap this benefit." With this focus on features (what the system does), it's no surprise that often security is described in the same way: we need a login page; we must have a fraud detection module; there should be logging.

Security experts John Wilander and Jens Gustavsson researched how security was described and specified. They studied a selection of major software initiatives that were

financed through public funding. When security was mentioned, they found that 78% could be directly classified as security features.⁵

Of course, there exist security features that add value, both visible and invisible. A visible example can be a high-quality authentication mechanism that allows customers to trust that their access and communications are safe.⁶ The problem is that describing security through features often misses the point. Let's try to phrase a security story to see how you can turn from a *feature focus* to a *concern focus*.

Let's look at an example of user authentication at a photo-storing website. If you try to squeeze this into the format of a function-focused user story, you might end up with something like this: "As a user, I want a login page so that I can access my uploaded pictures."

Although phrased as a *feature*, most probably the stakeholder is airing a *concern* about security. If you implement only this functionality, then you've met the objectives of the login page story. But the mere existence of a login page as such doesn't provide the security you're after. It might seem obvious that nobody really wants a login page like this, but we've seen this kind of feature-focused user story about security many times.

Imagine that you and your team implement a login page. After logging in, the user is redirected to a listing of their pictures, and among them is a really-embarrassing-pose.jpg. The user can click a link to get a download of the picture as well. To complicate things, imagine further that another user happens to have the direct download link and is also able to download that embarrassing photo (figure 1.1). How does that feel? You have implemented the story, because you have a login page with the described functionality, but you've subtly missed the point, have you not?

Taking a step back, we realize that the purpose of the story wasn't the login page as such. The purpose was rather that only the owner of the pictures should be able to see their pictures and download them—no one else. The login page is just there to uphold that rule. There should be no way for a user to get to the pictures without going through the login page.

You can now propose a better phrasing for the story: "As a user, I want access to my uploaded pictures to pass through a login page so that my pictures stay confidential." This phrasing better catches the concern the stakeholder was airing when initially talking about the login page.

An even better phrasing is not to mention the login page at all: "As a user, I want all access to my uploaded pictures to be protected by authentication so that my pictures stay confidential."

The point of the user story wasn't to have a security feature, it was to address a security concern; in this case, a concern about *confidentiality* (keeping things secret). The tricky part here is that when implementing such a story, it doesn't suffice to change the code along one path to the pictures. Instead, *all* paths leading to the pictures must be guarded, and it's enough to miss just one of them for the concern to not be met.

⁵ See Wilander, J., and Gustavsson, J., "Security Requirements—A Field Study of Current Practice," http://johnwilander.se/research_publications/paper_sreis2005_wilander_gustavsson.pdf.

⁶ The Swedish certificate-based authentication system BankID is one example that has become a de facto standard, beginning with financial institutions and governmental agencies but now encompassing lots of industries.

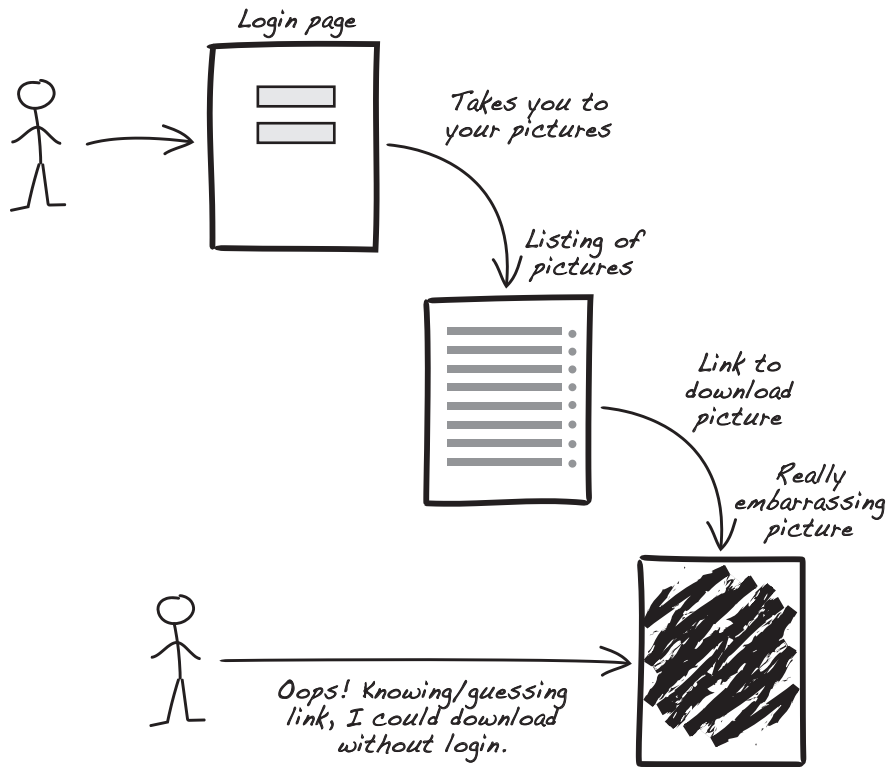


Figure 1.1 Having only a login page doesn't help much.

To get real security, you need to get away from thinking about security as a set of features. You must think about security as a cross-cutting concern—a concern that cuts across the functionality.

1.1.3 Categorizing security concerns: CIA-T

We've mentioned confidentiality as a security concern. But security is more than keeping things secret, and in this book, we'll talk about other aspects of security as well. To begin, let's provide some terminology around security concerns.

Classical information security usually talks about the security concern triad: confidentiality, integrity, and availability (or CIA as a mnemonic):

- **Confidentiality**—Most often associated with talking about security, is about keeping things secret that shouldn't be made known to the public. Your healthcare record is one of the best examples of confidential information.
- **Integrity**—Refers to when it's important that the information doesn't change or is only allowed to change in specific, authorized ways. An example of integrity is counting election results. Security in this context means that the votes haven't been manipulated.

- **Availability**—Means data is at hand in a timely manner. The fire department needs to know about the location of a fire, and they need that information immediately. If they get the location later, it might be too late, and the need for security can't be met.

All three factors might be important for any piece of data, but most often there's some kind of profile for the concern of how much you suffer from a breach. Take your health record, for example. If some data is revealed (breached confidentiality), you'll be irritated and angry. If there are errors in the data (breached integrity), things might get confused and dangerous. If the data isn't there when needed in the emergency room (breached availability), you might end up dead.

On the other hand, let's think about your bank record. If you can't see your balance (availability) when trying to pay your bills, it's irritating. If your balance is revealed publicly (confidentiality), you'll most probably be angry. But if your pension fund is suddenly wiped out (integrity), it's a catastrophe.

Later added to the CIA triad was the letter *T* for *traceability*, which captures the need for knowing who changed or accessed what data when. After some scandals, this became important in the financial sector and in healthcare. This kind of audit logging is also an important part of the European Union directive, **GDPR (General Data Protection Regulation)**, which went into effect in 2018. For example, GDPR specifies that when personal data is accessed, the access should be traced and saved to a persistent audit log. We'll refer to confidentiality, integrity, availability, and traceability in the rest of this book when we want to be a little bit more specific about what kind of security is at stake.

Focusing on security concerns instead of security features does a lot for the quality of the system, but it also puts developers in a difficult position: how do you ensure security in the software you write? It's hard to make sure there are no security mistakes anywhere. Ensuring this would require developers to actively think about security all the time while working. But there's another way—embed security into the way you work and the way you design.

1.2 Defining design

Writing software is by no means a trivial task. As a developer, you're required to have skills within a wide range of disciplines. You're expected to be knowledgeable in areas ranging from programming languages and algorithms to system architecture and agile methodologies. Although these software development disciplines span various fields of knowledge and can be quite different from one another, one term that keeps occurring when discussing almost all the different disciplines is *design*. But what do people mean when they use the word *design*?

Our view, in general, is that the word *design* is used quite loosely and takes on a different meaning depending on whom you talk to and in what context it's being used. We believe that design is an extremely important concept in software development, so important that we even put the word in the title of this book. As such, it's only appropriate to start by defining our view of the term *design* and how it's used throughout this

book. Understanding the meaning of the word will help you understand the discussions and concepts being conveyed in this book.

When developing software, you constantly make decisions on how to write the code that solves the problems at hand. You decide what syntax to use, what constructs and algorithms to apply, how to structure the code, and how to steer the flow of execution. If you're using an object-oriented approach, you'll make decisions on what your object model should look like and the interactions between the objects within that model. If you're applying a functional style of programming, you'll make decisions on what behavior to pass in as functions, making sure you're creating pure functions without side effects.⁷ All these decisions can be viewed as part of the design process.

When you write code, you pay careful attention to how to represent your business logic, which is the functionality that makes your software unique. You'll think about how you'll implement that logic and how to make it explicit and easy to maintain. If you're involved in activities around modeling your business domain, you'll spend a considerable amount of time evolving and refining your domain model and how it'll be represented in code. Even when you're implementing simple logic such as a straightforward conditional statement, you're making an active choice. For example, you might consider aspects such as readability or performance, and, based on your preferences, you'll make a decision on how you're going to write the code in that statement. You're drawing from your experience and knowledge to actively make choices appropriate to the software you're creating. These choices are part of what determines the design of the software.

As your codebase evolves, you'll put effort into structuring your code into packages or modules to make it more understandable and easier to work with, while at the same time achieving desirable properties like high cohesion and low coupling. You might apply techniques and concepts like the use of interfaces, the Dependency Inversion Principle,⁸ and immutability, while making sure you're not violating the Liskov Substitution Principle.⁹ You might also think about breaking out and isolating certain functionality within the code in order to make it more explicit or to allow it to be easily testable. What you're doing is writing and refactoring your code in order to give it a better design.

If your software is interacting with other software (say, for example, you're developing a service in a microservices architecture), then you're going to need to think about how to define the public API for your service in order for it to be cohesive and easy to consume and be versioned. You'll also need to consider how it'll interact with other services in order to be resilient and responsive, and to provide acceptable uptime. On a higher level, you're probably going to have to take into account that your service also needs to fit into the overall system architecture. You're making decisions that, albeit quite diverse, are all part of shaping the overall design of the software.

⁷ A pure function is a function that always returns the same result for a given argument and has no side effects.

⁸ See Martin, R. C., "The Dependency Inversion Principle," C++ Report 8 (May, 1996).

⁹ See Liskov, B., "Keynote Address—Data Abstraction and Hierarchy," OOPSLA '87 Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (1987).

All of the activities that we've discussed so far are related to writing code. We've stated that they are all part of the design process, but if you think about them for a moment, which would you say are design activities and which are not?

- Are API design and taking system architecture into account typical examples of design activities?
- Can domain modeling also count as a design activity?
- Is the choice between making the declaration of an object's field final or non-final a design activity?

If you ask 10 people what activities in software development count as design, then you're probably going to end up with 10 different answers. Many will probably answer that domain modeling, API design, applying design patterns, and system architecture are clearly examples of design activities, partly because this is the more traditional view of what design is. Whereas only a few, if any, will say that thinking hard about how to write an `if` statement or `for` loop qualifies as part of the software design process.

The answer to the question of which activities are design activities is that everything involved in software development is part of the design process. A system or piece of software won't reach a point of stable design (stable as in *functioning*, not as in having stopped evolving) until it has been written and put into production. That means that domain models, software modules, APIs, and design patterns are just as important to the design of the software as are field declarations, `if` statements, hash tables, and method declarations. All of these contribute to the stability of the design.

One thing that all these activities have in common is that they involve conscious decision-making. Any activity that involves active decision-making should be considered part of the software design process and can thus be referred to as design. This, in turn, means that design is the guiding principle for how a system is built and is applicable on all levels, from code to architecture.

NOTE Design is the guiding principle for how a system is built and is applicable on all levels, from code to architecture. It includes any activity that involves active decision-making.

In this section, you've learned how to view software design and what the word *design* means when used in this book. Next, we'll take a look at the traditional software security approach and some of its shortcomings.

1.3 The traditional approach to software security and its shortcomings

From our observation of the software industry, a common view when attempting to mitigate security vulnerabilities is that security should be a top priority when developing and writing code. Everyone involved in the process should be trained and experienced in software security. Let's refer to this view as the *traditional approach* to software security. This approach typically includes specific tasks and actions developers need to adhere to (figure 1.2).

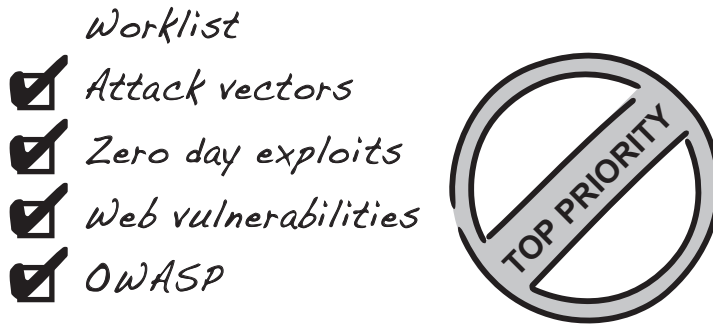


Figure 1.2 Traditionally, software security is viewed as explicit activities and concepts.

Developers should know about things like cross-site scripting (XSS) attacks, be aware of vulnerabilities in low-level protocols, and know the OWASP Top 10 like the backs of their hands.¹⁰ Testers should be trained in basic penetration testing techniques, and business domain experts should be capable of having discussions and making decisions concerning software security.

The weakness in this approach is that, for a number of reasons, it struggles to create software that's secure enough to withstand the harsh reality of production environments. If it had been successful, software security vulnerabilities wouldn't be as common as they are today, and we wouldn't see the same vulnerabilities responsible for massive security breaches over and over again. Let's take a closer look at some of the shortcomings of this approach to better understand why this approach struggles and why we think a different approach can be more successful.

As an example, say you have a simple domain object that represents a user in a typical web application, where the username is displayed on some page. The user object is quite simple and holds only an ID and a username. It's a simplified example but, in our experience, it's quite representative of code one might see. The implementation of the user object can be seen in the following listing.

Listing 1.1 Simple User class

```
public class User {
    private final Long id;
    private final String username;

    public User(final Long id, final String username) {
        this.id = id;
        this.username = username;
    }

    // ...
}
```

← Possible XSS vulnerability

¹⁰ See the Open Web Application Security Project (OWASP) Top 10, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

If you take a look at this representation of a user, you can see that there are possible security issues in this code. One issue is that because you're accepting any string value as a username, the username could be used for performing XSS attacks. An XSS attack occurs when an attacker uses a web application to send malicious code to a different user. The malicious code could, for example, be in the form of client-side JavaScript. If the attacker enters something like `<script>alert(42);</script>` as the username when registering for the service, later, when the user's username is displayed on some web page in the application, it could lead to an alert box being displayed in the browser showing the number 42.¹¹

If you want to mitigate this security vulnerability using the traditional approach, you could introduce explicit, security-focused input data validation. The data validation could, for example, be implemented as web application filters that validate all posted form data in the web application and check that it doesn't contain any malicious XSS code. Or the validation could occur right in the domain class. If you chose to introduce input validation in the User class, it could look something like that shown in the following listing.

Listing 1.2 User class with input validation

```
import static com.example.xss.ValidationUtils.validateForXSS;
import static org.apache.commons.lang3.Validate.notNull;

public class User {
    private final Long id;
    private final String username;

    public User(final Long id, final String username) {
        notNull(id);
        notNull(username);
        this.id = notNull(id);
        this.username = validateForXSS(username);
    }

    // ...
}
```

Checks that parameters aren't null

Validates input with an (imaginary) external library, ValidationUtils

You can see in the listing how you're pulling in an (imaginary) security library that provides functionality to validate a string for possible XSS attacks. You also decided to check that none of the constructor parameters are null to further improve the validation. This way of handling security in software is common, but it's also problematic for several reasons, some being:

- The developer needs to explicitly think about security vulnerabilities, while at the same time focusing on solving business functionality.
- It requires every developer to be a security expert.
- It assumes that the person writing the code can think of every potential vulnerability that might occur now or in the future.

¹¹ In a real attack, the executed script would most likely perform something a bit more evil than simply showing this number!

Let's take a look at each one of these issues and see why they're problematic.

1.3.1 **Explicitly thinking about security**

The first issue, thinking explicitly about security, is problematic, because when you as a developer are writing code, your main focus will always be the functionality you're trying to implement. Saying that you also need to actively think about security while coding is going to conflict with that focus. When that conflict occurs, security will always come in second to the priority of the business functionality. Security always gets a lower priority for a couple of reasons, and we'll look into those in more depth in section 1.4.2.

1.3.2 **Everyone is a security expert**

The next issue, requiring every developer to be a security expert, is also problematic because not everyone can be or wants to be such an expert, in the same way as everyone can't be an expert on JVM performance or UX design. And if the developers aren't highly skilled in security, then the software they create is only going to reflect the level of security that the developers are capable of.

Perhaps sometime in the future, all developers will need to have a thorough understanding of software security similar to the more or less mandatory knowledge nowadays of how to write good unit tests. But this isn't what the current state in the industry looks like, so it's somewhat of an unrealistic expectation.

1.3.3 **Knowing all and the unknowable**

Even if you have a team of security experts writing your software, you'd still face the fact that you can only write countermeasures for the vulnerabilities that you already know about. Not only do you need to know a lot about the many different types of attack vectors that you're familiar with, but you also need to know about the attacks that you currently are unaware of. You need to know the unknowable, so to speak. Once you realize this dilemma, it becomes obvious that the third issue also has its shortcomings in creating secure software.

The approach of creating secure software by making security the top priority has been around for as long as anyone can remember, and we've all tried it. Sometimes it has gotten the job done, but many times we've felt that there was something missing and that there should be a better way of creating secure software. We believe that software design is the enabler for successfully creating truly secure code. And by focusing on design, you avoid many of the shortcomings posed by the approach we've discussed in this section.

1.4 **Driving security through design**

We're not arguing that security isn't important or that you don't have to be aware of security when developing software. But we believe that, instead of adhering to the traditional approach to software security, there's an alternative approach that achieves the same or even better results when it comes to how secure your finished software will be (figure 1.3).

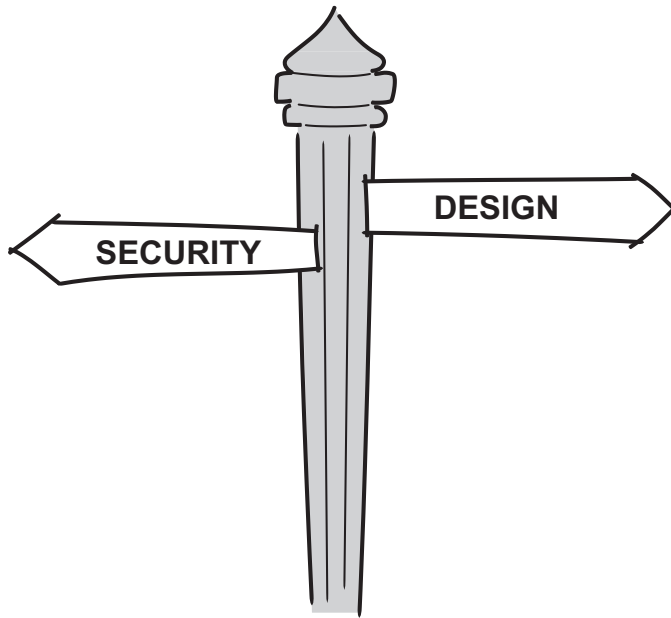


Figure 1.3 A focus on design rather than on security avoids issues with the traditional approach to security.

Rather than having security be one of the main focuses when you're developing software, you can choose to focus on software design instead—focusing on design in the sense that you're always aiming toward the highest possible standards with what you create. By shifting the focus to design, you'll be able to achieve a high degree of software security without the need to constantly and explicitly think about security.

1.4.1 Making the user secure by design

Let's go back to the example of the User class from the previous section and see how you'd approach it instead by focusing on good design. First, you'll discuss with your domain experts what the meaning of a username is in the context of the current application (figure 1.4).

You twist and turn on the concept and finally come to the conclusion that a username can only contain the characters [A-Za-z0-9_-] and must be at least 4 characters long, but no longer than 40. This is because that's what's considered to be a normal username in the application you're creating. You're not excluding characters like < or > because they might be part of an XSS attack in the event of the username being rendered in a web browser. Rather, you address the question, "In this context, what's a username supposed to look like?" In this case, you decide < or > isn't part of a valid username and shouldn't be included.

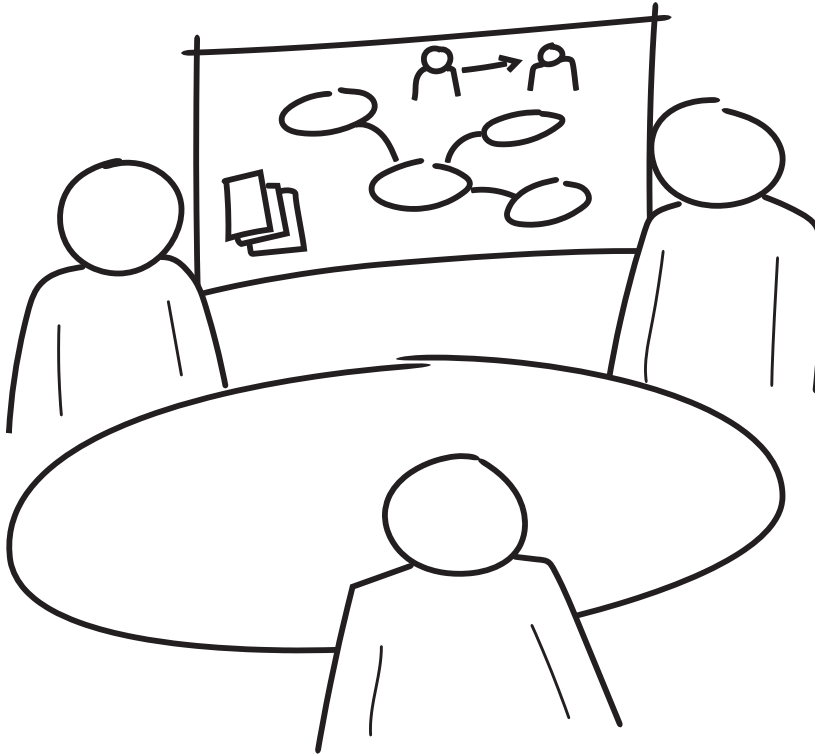


Figure 1.4 Exploring concepts with domain experts to gain deeper insight into the domain

This little exploration exercise with your domain experts has given you a deeper insight into the current domain that, in turn, allows you to create a more precise definition of a username. The following listing shows the new `User` class.

Listing 1.3 User class with domain constraints

```
import static org.apache.commons.lang3.Validate.*;

public class User {
    private static final int USERNAME_MINIMUM_LENGTH = 4;
    private static final int USERNAME_MAXIMUM_LENGTH = 40;
    private static final String USERNAME_VALID_CHARACTERS =
        "[A-Za-z0-9_-]+";

    private final Long id;
    private final String username;

    public User(final Long id, final String username) {
        notNull(id);
        notBlank(username);
    }
}
```



```

        final String trimmed = username.trim();
        inclusiveBetween(USERNAME_MINIMUM_LENGTH,
                        USERNAME_MAXIMUM_LENGTH,
                        trimmed.length());
        matchesPattern(trimmed,
                        USERNAME_VALID_CHARACTERS,
                        "Allowed characters are: %s",
                        USERNAME_VALID_CHARACTERS);

        this.id = id;
        this.username = trimmed;
    }

    // ...
}

```

Using domain invariants validates input at the time of creation.

Looking at the User class now, you can see that there's a lot of logic concerning the username of a user. This, together with the fact that you discussed it extensively with your domain experts, is a sign that the username should be represented explicitly in the domain model. That's partly because it seems to be an important concept, but also because extracting the logic would follow the principle of high cohesion.

With that insight, you can go ahead and extract the logic into its own Username class that encapsulates all the knowledge about a username. The new class also enforces all domain rules at the time of creation. This new object is called a *domain primitive* (you'll learn more about them in chapter 5). The following listing shows what your User class will look like once you've extracted the new Username class.

Listing 1.4 User class with domain value object

```

import static org.apache.commons.lang3.Validate.*;

public class Username {
    private static final int MINIMUM_LENGTH = 4;
    private static final int MAXIMUM_LENGTH = 40;
    private static final String VALID_CHARACTERS = "[A-Za-z0-9_-]+";

    private final String value;

    public Username(final String value) {
        notBlank(value);

        final String trimmed = value.trim();
        inclusiveBetween(MINIMUM_LENGTH,
                        MAXIMUM_LENGTH,
                        trimmed.length());
        matchesPattern(trimmed,
                        VALID_CHARACTERS,
                        "Allowed characters are: %s", VALID_CHARACTERS);
        this.value = trimmed;
    }
}

```

The value object that upholds the domain invariants for a username

```

    public String value() {
        return value;
    }
}

public class User {
    private final Long id;
    private final Username username;

    public User(final Long id, final Username username) {
        this.id = notNull(id);
        this.username = notNull(username);
    }

    // ...
}

```

The User object now uses the Username domain primitive, knowing that a username is always valid if one exists.

By focusing on design, you were able to find out more about the details surrounding a user and a username. This, in turn, let you create a more precise domain model. You also paid close attention to when concepts within the current domain became so important that they should be extracted into their own objects. In the end, you gained a deeper knowledge about your domain and, at the same time, protected yourself against the XSS vulnerability we discussed earlier; attempting to input `<script>alert(42);</script>` as a username becomes impossible because it's not a valid username anymore. And you haven't even started to think about security yet! If you were to consider security in your design choices, then you could probably tighten the restrictions on a username even more, hardening the code further, but still keeping the focus on good design.

NOTE A strong design focus lets you create code that's more secure when compared to the traditional approach to software security.

So far, you've learned about the shortcomings of the traditional approach, and you've seen how to use design to your advantage to create secure software. Some of the concepts that we've briefly touched on in this section will be explained in detail in chapter 3. There, you'll learn core concepts about Domain-Driven Design relevant for this book. Then, in chapters 4 and 5, we'll explain fundamental code constructs that promote security. Now, let's take a look at the advantages of driving security through design and why we believe this approach succeeds better than the traditional approach to software security.

1.4.2 **The advantages of the design approach**

In the simple User example, we showed you how to use design to drive security in your development process. We also stated that by focusing on design, you can achieve a level of software security that's on par with, or even better than, the traditional approach. But on what grounds do we make the claim that this approach succeeds better than the traditional one?

We believe that if the main focus when developing software centers on design, security can become a natural part of the development process instead of being perceived as a forced requirement. We also believe that this overcomes or avoids many of the shortcomings of the traditional approach and that it brings its own advantages. The main reasons for this follow:

- Software design is central to the interest and competence of most developers, which makes secure by design concepts easy to adapt.
- By focusing on design, business and security concerns gain equal priority in the view of both business experts and developers.
- By choosing good design constructs, nonsecurity experts are able to write secure code.
- By focusing on the domain, many security bugs are solved implicitly.

Let's take a closer look at the reasoning behind these advantages and why we believe the design approach succeeds better than the traditional approach.

DESIGN IS A NATURAL PART OF SOFTWARE DEVELOPMENT

As software developers, you're taught from early on the importance of good design. You study it and you take pride in creating good designs that serve their purpose well. This makes design a natural part of creating software.

Many developers feel like it's hard to understand all the details around intricate software vulnerabilities, classifying themselves as people who don't do security—security is something that's best left to someone else. But because most developers understand and appreciate software design, if you can use design to achieve security, then suddenly everyone can create secure software.

When you focus on design, security becomes the concern and interest of everyone, not only the experts. It also means that there's no longer a conflict between business functionality and security concerns because the distinction between them no longer exists. This reduces the cognitive load on the developer and avoids one of the shortcomings of the traditional approach.

BUSINESS CONCERNS AND SECURITY CONCERNS BECOME OF EQUAL PRIORITY

One issue with the traditional approach is that it treats security as a separate activity. This forces security to compete with all the other important aspects you're trying to address, such as business functionality, scalability, testability, maintainability, and so on. Security-related tasks are added to the backlog and prioritized against everything else that needs to be done.

When you determine the priority of the different tasks, there's nothing that says security should automatically get the fast lane in the backlog. But what we often see is that security tends to consistently get too low of a priority. Here are some of the reasons for this:

- Security isn't well understood by either the business side or the development side of the organization.

- Developers tend to think security isn't their concern because of the reasons we discussed earlier.
- Even if security is understood, it's easy to think of it as less important than user features, and something that can therefore be added at a later time.

The caveat with the notion of adding security later is that it might not be possible if the security aspects needed imply a fundamentally different design. This is similar to why it's usually hard or impossible to add scalability or statelessness late in the software cycle.

By focusing on design and domain knowledge (as you did in the previous example with the User), you're removing several of the situations where it's necessary to prioritize security against other tasks in the backlog. It's no longer a question about whether to implement a security feature or a business feature. Now it's about implementing functionality relevant to your domain.

Finally, the design focus also makes security more accessible to all stakeholders, not only the experts. This is because it's easier to reason about, see the value in, and prioritize tasks that are related to your domain rather than specific security vulnerabilities.

NON-SECURITY EXPERTS NATURALLY WRITE SECURE CODE

Another interesting benefit of using a secure by design approach is that non-security experts can now naturally write secure code. This isn't because they consider attack vectors and how malicious data might affect the system, but rather because the design implicitly avoids insecure constructs. To illustrate this, consider the `Username` class in listing 1.4, where invariants ensure only valid usernames are accepted. How do we justify using this complex type instead of a simple string?

As it turns out, when talking to domain experts, most developers realize the importance of representing business concepts as precisely as possible. A username isn't an unbounded random sequence of characters; it's a well-defined concept with a precise meaning and purpose in the domain. Representing this by the standard `String` class isn't only a poor design decision, it's completely wrong—an insight that makes preciseness and correctness the natural choice for any developer, regardless of interest in security or experience level.

DOMAIN FOCUS SOLVES SECURITY BUGS IMPLICITLY

Security issues are often perceived as scary and complicated, but when using the design approach, the complexity suddenly disappears. This is primarily because the distinction between security bugs and ordinary bugs is erased when focus is placed on the domain rather than on which countermeasure to use.

If you look at `Username` in listing 1.4, the main reason for applying strict invariants isn't to protect a username from injection attacks, but rather to ensure the true meaning of a username is captured. As a consequence, every malicious input not satisfying the definition is rejected, and a username becomes secure by focusing on the domain rather than by thinking about security. The domain focus reduces the risk of security bugs in your code and, in some cases, it can also protect you against security bugs in third-party code.

1.4.3 Staying eclectic

As we mentioned earlier, if you complement your focus on design with a more traditional and explicit security awareness, then the resulting code becomes even more secure. This is an important note to point out, because the design focus gives you a high level of security but never covers all the security needs a system has (nor is that the intention). There's always a need to perform tasks like penetration testing and to actively think about specific attack vectors and vulnerabilities when creating software systems.¹² Even if the domain focus makes Username in the example secure, you still have to remember to perform proper output encoding when displaying it on a web page. By keeping the focus on design and at the same time taking an eclectic approach to software security, you can create truly secure software.

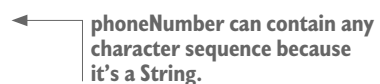
We've gone through quite a lot of material so far, but we believe it's important to understand the *why* before looking at the *how*. You've learned the meaning of design and the fundamental thinking behind the idea that a strong design focus can drive security in software development. You've also seen a simple example of how this can work. In the next section, we'll take a look at a slightly more complex scenario to give you another example of how design can improve security.

1.5 Dealing with strings, XML, and a billion laughs

When designing software, you're often faced with the decision of how to represent data. Unfortunately, there's a tendency towards using data types that are too generic for the purpose. For example, representing a phone number as a string can seem convenient at first, but from a security perspective, it's devastating, because a string can represent almost any kind of data—not just what you'd expect. Still, developers tend to favor strings, and often the protection against invalid input is enforced by *name typing* as seen in the next listing. The method register expects a phone number, but the argument is a String, which means it could be anything!

Listing 1.5 A String argument protected by name typing

```
public void register(String phoneNumber) {  
    ...  
}
```



Obviously, preventing invalid input this way doesn't work. The solution is to use strict domain types with rules, as you saw with User and Username earlier. But using strict types is only half of the story.

If you dissect Username, you see that the validation logic in the constructor contains a notBlank and a length check before applying the regular expression. This turns out to be

¹² We'll discuss some of the other aspects important for software security in more depth in chapter 14.

extremely important from a security perspective, and we'll further discuss why this is in chapter 4. So, for now, accept that validation should be executed in the following order:

- *Length check*—Is the input length within the expected boundaries?
- *Lexical content check*—Does the input contain the right characters and encoding?
- *Syntax check*—Is the input format right?

Up to this point, we've only touched on simple examples using validation, but that doesn't mean validation can't be used in more complex situations as well. To illustrate, we'll walk you through an example where you'll learn how to process XML securely. This seems quite remote from the previous examples, but when applying the same validation principles, you'll see that the parsing complexity is reduced to an ordinary input validation problem. So let's proceed with some XML.

1.5.1 *Extensible Markup Language (XML)*

XML is similar to a string in the sense that it can represent almost any kind of data.¹³ Because of this, XML is often used as an intermediate data representation when communicating between systems. Unfortunately, not many realize there's a lot more to XML than just representing data on a normalized form.

XML is really a complete language derived from SGML (Standard Generalized Markup Language), which means there are probably features supported by XML that most developers don't care about. Consequently, many security weaknesses are introduced in software because of how XML is used. And to illustrate, we'll use the *Billion Laughs attack* (which exploits the expandability property of XML entities during the parsing process) as a foundation when learning how to process XML securely. But before we dive into details, let's take a quick refresher on how internal XML entities work.

1.5.2 *Internal XML entities in a nutshell*

Internal XML entities are powerful constructs that allow you to create simple abbreviations in XML. They're defined in the Document Type Definition (DTD) and written in the form `<!ENTITY name "value">`. The following listing shows a simple example of an entity that's an abbreviation of *Secure by Design*.

Listing 1.6 Defining an entity and referencing it in XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE example [
<!ELEMENT example (#PCDATA)>
<!ENTITY title "Secure by Design">
]>
<example>&title;</example>
```

References the title entity

When the XML parser encounters the `title` entity, it expands the abbreviation and replaces it with the value found in the DTD. This, in turn, leads to a rich XML block without abbreviations, as seen in the next listing.

¹³ For more, see W3C, <https://www.w3.org/XML/>.

behave differently. To get a solid foundation, a good starting point is to use external resources such as OWASP (the Open Web Application Security Project) as a guide.

Listing 1.9 provides an example of a parser configuration based on OWASP’s recommendations that attempts to avoid entity expansion.¹⁴ The selected features are quite invasive because almost everything regarding entities is disabled. For example, disallowing doctype does indeed make it difficult to do an entity attack, but at the same time, it affects overall usability. In these situations, security concerns are often compared against business needs, and if it’s decided to weaken the configuration, it’s important to understand what the risks are.

Listing 1.9 XML parser configuration suggested by OWASP

```
import static javax.xml.XMLConstants.FEATURE_SECURE_PROCESSING;

public final class XMLParser {
    static final String DISALLOW_DOCTYPE =
        "http://apache.org/xml/features/disallow-doctype-decl";
    static final String ALLOW_EXT_GEN_ENTITIES =
        "http://xml.org/sax/features/external-general-entities";
    static final String ALLOW_EXT_PARAM_ENTITIES =
        "http://xml.org/sax/features/external-parameter-entities";
    static final String ALLOW_EXTERNAL_DTD =
        "http://apache.org/xml/features/nonvalidating/load-external-dtd";

    public static Document parse(final InputStream input)
        throws SAXException, IOException {
        try {
            final DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

            factory.setExpandEntityReferences(false);
            factory.setFeature(FEATURE_SECURE_PROCESSING, true);
            factory.setFeature(DISALLOW_DOCTYPE, true);
            factory.setFeature(ALLOW_EXT_GEN_ENTITIES, false);
            factory.setFeature(ALLOW_EXT_PARAM_ENTITIES, false);
            factory.setFeature(ALLOW_EXTERNAL_DTD, false);

            return factory.newDocumentBuilder().parse(input);
        } catch (ParserConfigurationException e) {
            throw new IllegalStateException("Configuration Error", e);
        }
    }
}
```

Instructs the parser to process XML securely →

← **Disables entity reference expansion**

← **Disallows DTDs in XML**

← **Disallows external general entities**

← **Disallows external parameter entities**

← **Disallows loading external DTDs**

Even though relying on the parser configuration is recommended, it feels as if there’s a lot of risk to it. For example, what happens if the underlying parser implementation changes or a feature is forgotten? These concerns are valid, and to address them, we recommend applying another layer of security—design.

¹⁴ See the “XML External Entity (XXE) Prevention Cheat Sheet,” <https://github.com/OWASP/Cheat-SheetSeries/blob/master/cheatsheets/>.

TIP To detect entity expansion, add a test in your build pipeline with a recursive entity definition. If the entity is expanded and the XML is accepted, then the test should fail because the parser might be vulnerable to expansion attacks.

1.5.5 Applying a design mindset

Before approaching the Billion Laughs problem from a design perspective, we think it's important to let go of the idea that the root cause lies in how entities are expanded. This is because the expansion is in accordance with the XML specification and not the result of a faulty parser implementation. This implies that the problem shouldn't be treated as a structural problem of XML, but rather as an input validation problem in the receiving system. This, in turn, means that a malicious XML block (such as the Billion Laughs XML) must be rejected by the receiving system without parsing it—this certainly sounds appealing, but is it a viable solution? It definitely is, and the answer lies in the second step of the validation order presented earlier—to run a lexical content scan. It can seem like a complex operation at first, but running a lexical content scan isn't that difficult. It's simply the process of converting a stream of characters into a sequence of tokens without analyzing their order or meaning (because that's the job of the parser).

There are many ways to implement a lexical content scan. In listing 1.10, you see an example of using a SAX parser (Simple API for XML) to scan XML. It's somewhat counterintuitive to use a parser as a lexical content scanner, but a SAX parser is in fact quite suitable because it emits an event for each token that's identified in the data stream. These events can then be used to analyze contents, which would be done if using it as a parser or, as in this example, to reject XML with entities. The `startEntity` method of the `ElementHandler` in the example achieves this by throwing an exception to abort the scan as soon as an entity is detected.

Listing 1.10 Simple lexical scanner that detects entities

```
import static org.apache.commons.lang3.Validate.notNull;

public class LexicalScanner {
    private static final String LEXICAL_HANDLER =
        "http://xml.org/sax/properties/lexical-handler";

    public static boolean isValid(final InputStream data)
        throws Exception {
        notNull(data);

        final SAXParser saxParser = SAXParserFactory
            .newInstance().newSAXParser();
        final ElementHandler handler = new ElementHandler();

        saxParser.getXMLReader().setProperty(LEXICAL_HANDLER, handler);
        try {
            saxParser.parse(data, handler);
            return true;
        }
    }
}
```

Creates a SAX parser →

← **Creates a lexical element handler to detect entities**

→ **Registers the handler to listen to lexical events**

← **Scans the XML for entities**

```

    }
    catch(IllegalArgumentException e) {
        return false;
    }
}

public static final class ElementHandler extends
                                org.xml.sax.ext.DefaultHandler2 {
    @Override
    public void startEntity(final String name) throws SAXException {
        throw new IllegalArgumentException("Entities are illegal");
    }
}

```

← **Aborts the scan if an entity is found**

The scanner does indeed meet the expectations, but the main objective of the lexical scan is greater than just rejecting entities. A lexical scan should also ensure that all required elements exist in the XML; otherwise, it doesn't make sense to parse it. To illustrate, assume there's a customer object with exactly one phone number, email, and address represented in XML, as shown in listing 1.11. The phone number and address are required elements, whereas the email is optional. The only time it makes sense to pass the customer XML to the parser is when it contains all the required elements. All other element combinations are invalid from a business perspective and should be rejected by the scan—similar to how invalid input was rejected in the Username example earlier.

Listing 1.11 XML example representing a customer object

```

<customer>
  <phone>212-111-2222</phone>
  <email>jane.doe@example.com</email>
  <address>
    <street>Fifth Ave</street>
    <city>New York</city>
    <country>USA</country>
  </address>
</customer>

```

To ensure the customer XML contains all required elements, a richer design of the `ElementHandler` is needed. But before diving into details, it's important to remember that a lexical scan only cares about the existence of elements, not the meaning, the order, or if the elements exist multiple times. This allows for a structurally incorrect customer XML block (for example, with multiple phone numbers or address elements) to pass a lexical scan. Even though this seems like a flaw, the behavior is exactly as intended. Because as soon as semantic analysis is added to a lexical scan, it turns into a parsing process, and that brings everything back to square one.

With this in mind, let's turn back to the updated `ElementHandler` in listing 1.12. The implementation shows a couple of interesting details worth pointing out. First, all of the required elements are stored in a collection that's consulted in the `startElement` method each time an element is found. This looks straightforward, but there's a subtle trick to this that's easy to miss. Because the lexical scan only cares about the existence

of elements, it needs a mechanism to determine if all required elements are present at least once in the XML. This is achieved by trying to remove each detected element from the required elements collection. It doesn't matter if a detected element is required or not, it only matters that it doesn't exist in the collection after the remove operation. This is because when reaching the end of the data stream, the scanner needs to ensure that all required elements have been found, which is verified by checking that the collection is empty in the endDocument method.

Listing 1.12 Element handler with required elements check

```
import static org.apache.commons.lang3.Validate.isTrue;
import static org.apache.commons.lang3.Validate.notNull;

private static final class ElementHandler extends
    org.xml.sax.ext.DefaultHandler2 {

    private final Set<String> requiredElements = new HashSet<>();

    public ElementHandler() {
        requiredElements.add("customer");
        requiredElements.add("phone");
        requiredElements.add("address");
        requiredElements.add("street");
        requiredElements.add("city");
        requiredElements.add("country");
    }

    @Override
    public void startElement(final String uri,
        final String localName,
        final String name,
        final Attributes attributes)
        throws SAXException {
        notNull(name);
        final String element = name.toLowerCase();
        requiredElements.remove(element);
        isTrue(!requiredElements.contains(element));

        @Override
        public void endDocument() throws SAXException {
            isTrue(requiredElements.isEmpty());
        }

        @Override
        public void startEntity(final String name) throws SAXException {
            throw new IllegalArgumentException("Entities are illegal");
        }
    }
}
```

All required elements

Normalizes the element name to lowercase

Removes the element from the collection if and only if it exists

Ensures the element doesn't exist in the collection

Verifies that all required elements have been encountered

The second detail worth mentioning is the choice of using a liberal scanning strategy to ignore all nonrequired elements. This can seem like a potential weakness because it accepts customer XML that fails the parser requirements, but the choice is carefully made. When communicating between systems, Postel's Law and the Tolerant Reader pattern state that

an implementation should be liberal when receiving data and conservative when sending data.¹⁵ This makes system integration less painful because changes to data fields ignored by the receiving system becomes seamless to the overall integration. As a result, choosing to ignore all nonrequired elements makes the lexical scan resilient against less important data changes, such as updating an optional element in the customer XML.

This certainly makes it difficult to inject a Billion Laughs XML block, but what if entities are required? Wouldn't that render the lexical scan obsolete? Or is there a way to accept entities and prevent expansion attacks at the same time? There is, but to see how, we need to approach this from a different angle.

1.5.6 *Applying operational constraints*

Both the lexical scan and the parser configuration address expansion attacks by blindly rejecting XML with entities, regardless if they're malicious or not. This has the downside of working only when entities are illegal. All other situations call for a different solution. It's therefore interesting to revisit the Billion Laughs XML and try to understand where the real danger lies.

The primary suspect is the entity expansion, but that in itself isn't what makes entities unsafe to parse. Instead, it's the resulting memory footprint that's dangerous. This implies that parsing XML with entities isn't dangerous per se, but rather the actual size of the resulting XML. A viable solution is therefore to allow entity expansion but with operational constraints on the parser process (such as memory limits or quotas) to prevent runaway resource consumption.

Choosing this approach, however, doesn't automatically protect against resource depletion. Even if a single parser process is prevented from consuming too much resources (because it's killed when exceeding the limits), running processes in parallel can result in a similar situation as with the Billion Laughs attack. For example, assume there are parser processes running in parallel, where each process consumes the maximum amount of resources. The total amount of resources used is then proportional to the number of processes, which creates a significant resource consumption footprint. Consequently, any other part of the system that relies on the same resources (for example, CPU or memory) will be affected. This calls for a design where parsing is done in isolation because it reduces the risk of cascading failures. We'll elaborate more on this when discussing bulkheads in chapter 9.

Relying on operational constraints does indeed seem like a viable solution, but it doesn't render the use of a lexical scan or parser configuration obsolete. In fact, choosing a design where all strategies are applied makes the system even more resilient against expansion attacks, which brings us to the next topic—achieving security in depth.

1.5.7 *Achieving security in depth*

Most developers have a tendency to address entity expansion attacks using parser configuration only. This isn't flawed per se, but it's like building a fence around a house

¹⁵ See RFC 760 for Postel's Law at <https://tools.ietf.org/html/rfc760>, and for the Tolerant Reader pattern, see <https://martinfowler.com/bliki/TolerantReader.html>.

without locking the doors. No one is able to enter the house as long as the fence holds, but if it's breached, access is granted. This really isn't desirable. The obvious solution is to lock the doors and perhaps add an alarm on the inside. This is what security in depth is all about. With multiple layers of security, it becomes a lot harder for an attack to be successful even if a single protection mechanism is breached.¹⁶

If we look at the design for dealing with the Billion Laughs XML and correlate it to the house metaphor, it becomes easy to see how it achieves security in depth. By configuring the parser, a strong fence is built around the house. Sometimes this is too strict, and you can't reject all types of entities. In those situations, it's important not to remove the entire fence, but rather to understand what type of entities are needed. It might be possible to weaken the configuration to only accept certain types of entities (for example, only internal entities), which isn't perfect, but it's still a fence around the house.

The lexical scan process made sure that only XML with required elements was passed to the parser. This is similar to only letting people with keys into the house. That way, the set of XML blocks that need parsing is significantly reduced to those that might meet the business requirements. In turn, this makes it a lot harder to exploit the parser because the attack vector is now reduced to XML blocks with required elements. But what about entities? What if you need to accept them?

This is where the last layer of protection comes in. By applying operational constraints on the parser process, it's acceptable to weaken the lexical scan and pass XML with entities to the parser—similar to having a window open on the second floor. The operational constraints then make sure the parsing process never consumes too much resources—like a watchdog inside the house.

All in all, by applying parser configuration, lexical scan, and operational constraints together, it becomes significantly harder to do an expansion attack. And this is what secure by design is all about: using design as the primary tool and mindset for creating secure software. In the next chapter, we'll dive into a real-world case story that shows how brittle design and a lack of domain knowledge caused significant economic loss for a big global company, a situation that could have been avoided using secure by design principles.

Summary

- It's better to view security as a concern to be met than to view it as a set of features to implement.
- It's impractical to achieve security by keeping it at the top of your mind all the time while developing. A better way is to find design practices that guide you to more secure solutions.
- Any activity involving active decision-making should be considered part of the software design process and can thus be referred to as design.
- Design is the guiding principle for how a system is built and is applicable on all levels, from code to architecture.

¹⁶ See "Defense in Depth," <https://www.us-cert.gov/bsi/articles/knowledge/principles/defense-in-depth>.

- The traditional approach to software security struggles because it relies on the developer to explicitly think about security vulnerabilities while at the same time trying to focus on implementing business functionality. It requires every developer to be a security expert and assumes that the person writing the code can think of every potential vulnerability that can occur now or in the future.
- By shifting the focus to design, you're able to achieve a high degree of software security without the need to constantly and explicitly think about security.
- A strong design focus lets you create code that's more secure compared to the traditional approach to software security.
- Every XML parser is implicitly vulnerable to entity attacks because entities are part of the XML language.
- Using generic types to represent specific data is a potential door opener for security weaknesses.
- Choosing the XML parser configuration is difficult without understanding the underlying parser implementation.
- Secure by design promotes security in-depth by adding several layers of security.