CHAPTER 1

Spring Security IN ACTION

Laurențiu Spilcă

MANNING

Spring Security in Action

by Laurenţiu Spilcă

Chapter 1

Copyright 2020 Manning Publications
To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617298547

contents

1 Security today

This chapter covers

- What Spring Security is and what you can solve by using it
- What security is for a software application
- Why software security is essential and why you should care
- Common vulnerabilities that you'll encounter at the application level

Today more and more developers are becoming aware of security. It's not, unfortunately, a common practice to take responsibility for security from the beginning of the development of a software application. This attitude should change, and everyone involved in developing a software system should learn to consider it from the start.

Generally, as developers, we start by learning that the purpose of an application is to solve business cases. This purpose refers to something where data could be processed somehow, persisted, and eventually displayed to the user in a specific way as specified by some requirements. This overview of software development, which is somehow imposed from the early ages of development, has the unfortunate disadvantage of hiding practices that are also part of the process. While the application

works correctly from the user's perspective, and in the end, it does what the user expects in terms of functionalities, there are many aspects hidden in the final result.

Non-functional software qualities, such as performance, scalability, availability, and (of course) security, as well as others, may have an impact over time, from short to long term. If not taken into consideration early on, these qualities can affect in terms of profitability the owners of the application dramatically. Moreover, they could trigger failures in other systems (for example, by the unwilling participation in a distributed denial of service (DDoS) attack). The hidden aspect of non-functional requirements (the fact that it's much more challenging to see if they're missing or incomplete) makes them, however, more dangerous (see figure 1.1).

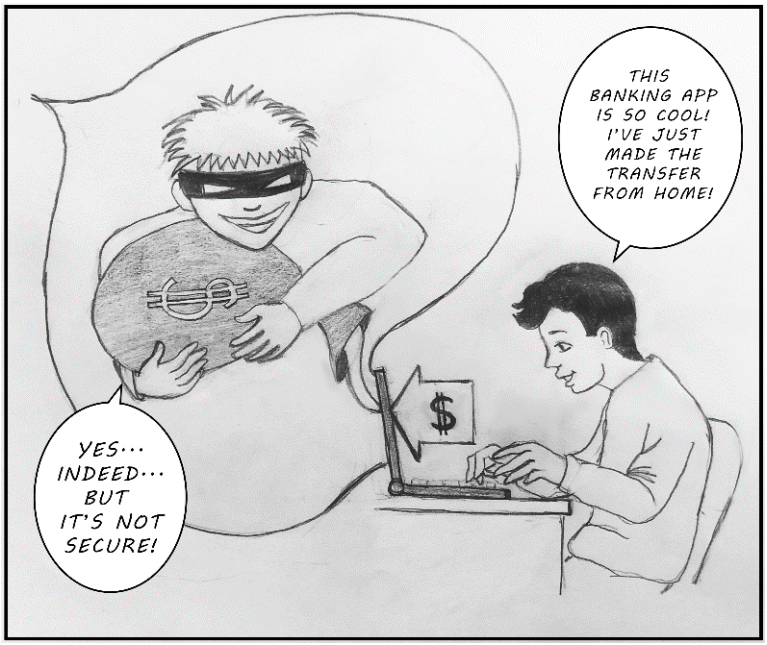

Figure 1.1 A user mainly thinks about the functional requirements. Sometimes you might see them aware also of performance—which is non-functional, but it's unusual unfortunately that they care about security. Non-functional requirements tend to be more transparent than functional ones.

You have multiple non-functional aspects to consider when working on a software system. In practice, all of these are important, and they need to be treated responsibly in the process of software development. In this book, we'll focus on one of them: security. You'll learn how to protect your application step by step using Spring Security.

But before starting, I'd like to make you aware of the following: depending on how much experience you have, you might or not find this chapter cumbersome. Don't worry too much if you don't understand absolutely all the aspects at the moment. For now, with this chapter, I want to show you a big picture of the security-related con-

cepts. Throughout the book, we'll work on practical examples, and where appropriate, I'll refer back to the description I give in this chapter. Where applicable, I'll also provide more details. Here and there, you'll find references to other materials (books, articles, documentation) that are useful for you to read on specific subjects.

1.1 Spring Security—the what and the why

In this section, we discuss the relationship between Spring Security and Spring. It's important first of all to understand the link between the two before starting to use them.

If you go to the official website, https://spring.io/projects/spring-security, they describe Spring Security as a powerful and highly customizable framework for authentication and access control. I'd say it's a framework that enormously simplifies baking security for Spring applications.

Spring Security is the primary choice for implementing application-level security in Spring applications. Generally, its purpose is to offer you a highly customizable way of implementing authentication, authorization, and protection against common attacks. Spring Security is open-source software released under the Apache 2.0 license. You can access the source code of the project on GitHub at https://github.com/spring-projects/spring-security/, and I highly recommend that you contribute to the project as well.

> **NOTE** You can use Spring Security for both standard web servlets as well as reactive applications. To use it, you need at least Java 8, although the examples in this book use Java 11, which is the latest long term supported version.

I can guess that if you opened this book, you work on Spring applications, and you're interested in securing them. Spring Security is probably the best choice for your cases. It became the de-facto choice in implementing the application-level security for Spring applications. Spring Security, however, doesn't automatically secure your application. It's not a kind of magic that guarantees a vulnerability-free app. Developers need to understand how to configure and customize Spring Security around the needs of the application. How to do this depends on many factors, from the functional requirements to the architecture.

Technically applying security with Spring Security in Spring applications is simple. You already implement Spring applications so you know that the framework's philosophy starts with the management of the Spring context. You define beans in the Spring context to allow the framework to manage them based on configurations you specify. And let me refer only to using annotations to make these configurations and leave behind the old-fashioned XML configuration style!

You can use annotations to instruct Spring what to do: expose endpoints, wrap methods in transactions, intercept methods in aspects, and so on. Also, you'd like to apply security configurations. This is where Spring Security comes into action. What you want is to use annotations, beans, and in general Spring-fashioned configuration style to define your application-level security. If you think of a Spring application, the behavior that you need to protect is defined by methods.

To think about application-level security, you can consider your home and the way you allow access to it. Do you place the key under the entrance rug? Do you even have a key for your front door? The same concept applies to applications, and Spring Security helps you develop this functionality. It's a puzzle that offers plenty of choices for building the exact image that describes your system. You can choose to leave it completely unsecured. Or you can decide not to allow everyone to enter your home.

The way you configure security could be simple, such as hiding your key under the rug, or it could be more complicated, such as choosing a variety of alarm systems, video cameras, and locks. In your applications, you have the option of doing the same. But as in real life, the more complexity you add, the more expensive it gets. In an application, this cost refers to the way security affects maintainability and performance.

But how do you use Spring Security with Spring applications? Generally, at the application level, one of the most encountered use cases refers to deciding whether an entity is allowed to perform an action or use a piece of data. Based on configurations you write, Spring Security components intercept the requests and make sure that who makes the requests has the permissions to access specific resources. The developer has to configure them in a way that they do precisely what's desired. If you mount an alarm system, it's you who should make sure it's also set up for the windows as well as for the doors. If you forget to set it up for the windows, it's not a fault of the alarm system that it didn't trigger when someone forced a window.

Other responsibilities that these components have also relate to the storing of data as well as transiting data between different parts of the systems. By intercepting the calls to these different parts, the components can act on the data. When the data is stored, these may apply encryption or hashing algorithms. The data encodings keep the data accessible only to privileged entities. In the Spring application, the developer has to add and configure a component to do this part of the job wherever it's needed. Spring Security provides a contract through which we know what the framework requires to be implemented, and we write the implementation according to the design of the application. We can say the same thing about transiting data.

Cases exist in which two components, communicating one with the other, can't trust themselves. How could the first know that the second one sent a specific message, and it wasn't someone else? Imagine you have a phone call with somebody to whom you have to give private information. How do you make sure that on the other side is indeed a valid individual with the right to get that data and not somebody else? For your application, this situation applies as well. Spring Security helps again with components and contracts that allow you to solve this in several ways. You have to know the part to configure and set it up in your system. This way, Spring Security intercepts the messages and makes sure to validate the communication before the application uses any kind of data sent or received.

As any part of a framework, one of its primary purposes is to allow you to write less code to implement the desired functionality. And this is also what Spring Security does. It completes Spring as a framework with helping us write less code to perform

one of the most critical aspects of an application—security. Spring Security provides predefined functionality to help you avoid writing boilerplate code or repeatedly writing the same logic from application to application. But it, as well, allows you to configure any of its components, providing great flexibility.

Short recap on this discussion:

- You use Spring Security to bake application-level security in your applications in the "Spring way". By this I mean, you'll use annotations, beans, Spring Expression Language (SpEL), and so on.
- Spring Security is a framework that allows you to build application-level security, not only a dependency you add such that your applications become magically secured.
- You have to know where, what and why to apply from Spring Security so protected your Spring applications. You'll learn all you need related to this aspect in this book.

Alternatives to Spring Security

This book is about Spring Security. But as with any other solution I always prefer to have a broad overview. Never forget to learn the alternatives that you have for any option. One of the things I've learned over time is that there's no general right or wrong. Everything is relative also applies here!

There aren't many alternatives to Spring Security when it comes to the security of a Spring application, but you could consider Apache Shiro (https://shiro.apache.org). It offers flexibility in configurations and is easy to integrate with Spring and Spring Boot applications. Apache Shiro makes, sometimes, a good alternative to the Spring Security approach.

If you've already worked with Spring Security, you'll find using Apache Shiro easy and comfortable to learn and use. It offers its own annotations and design for web applications based on HTTP filters, which are of great simplicity for web applications. Also, you can secure more than web applications with Shiro, from smaller command-line applications and mobile applications to large-scale enterprise applications. And even if simpler, it's powerful enough to use for a wide range of things from authentication and authorization to cryptography and session management.

However, Apache Shiro could be too "light" for the needs of your application. Spring Security is not only a hammer, but an entire set of tools. It offers you a larger scale of possibilities and is designed specifically for Spring applications. Moreover, it benefits from a larger community of active developers, and it's continuously enhanced.

1.2 What is software security?

Software systems today manage large amounts of data, out of which a significant part can be considered sensitive, especially given the current General Data Protection Regulations (GDPR) requirements. Any information that you, as a user, consider private is sensitive for your software application. Sensitive data could include inoffensive information such

as a phone number, email address, or identification number; although, we generally think more about data that's riskier to lose, such as your credit card details. The application should ensure that there's no chance for that information to be accessed, changed, or intercepted. No other parties than the users to whom they are intended should be able to interact in any way with it. Loosely defined, this is the meaning of security.

NOTE GDPR created a lot of buzz around the world after its introduction in 2018. It generally represents a set of European laws that refer to data protection and gives people more control over their private data. GDPR applies to the owners of systems having users in Europe. The owners of such applications risk significant penalties if they don't respect the regulations imposed.

Security should be applied in layers, with each layer requiring a different approach. Think about these layers as to a protected castle (figure 1.2). A hacker needs to bypass several obstacles to obtain the application's resources. The better security is applied at each layer, the least chance is that a bad intentioned individual manages to access data or do operations they aren't allowed to.

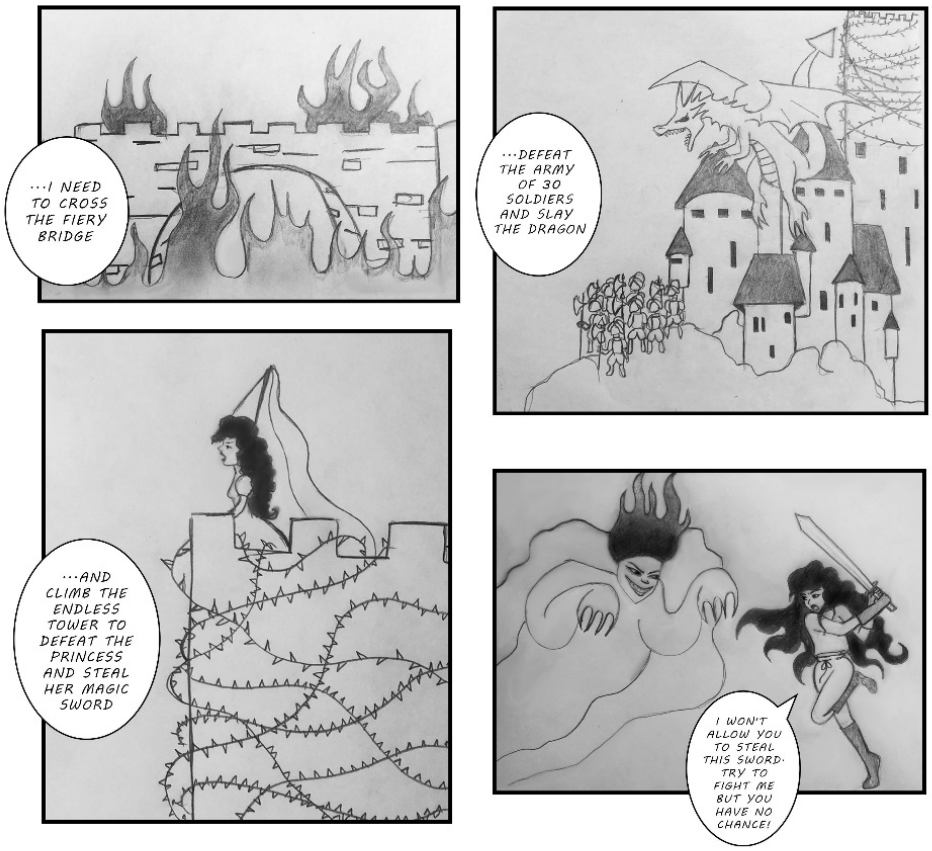

Figure 1.2 The Dark Wizard (a hacker) has to bypass multiple obstacles (security layers) that steal the Magic Sword (user resources) from the Princess.

Security is a complex subject. In the case of a software system, security doesn't apply only at the application level. For example, for networking, there are issues to be taken into consideration and specific practices to be used, while for the storage, it's another discussion. Similarly, a different philosophy applies in terms of deployment and so on. Spring Security is a framework that belongs to application-level security. In this section, you'll get a general picture of this security level and its implications.

Application-level security (figure 1.3) refers to everything that an application should do to protect the environment it executes in, as well as the data it processes and stores. Mind that this isn't only about the data affected and used by the application. An application might contain vulnerabilities that allow a malicious individual to affect the entire system!

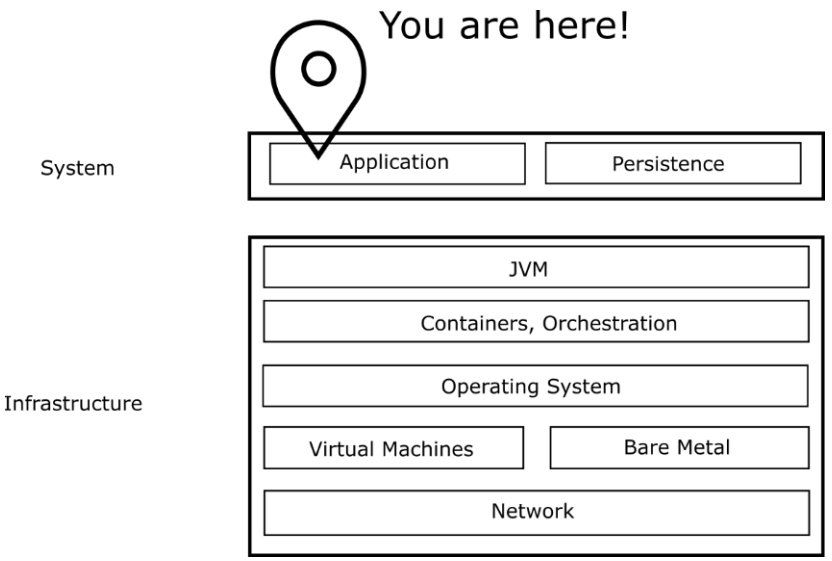

Figure 1.3 We apply security in layers. Each of the layers depends on those below them. In this book, we discuss Spring Security, which is a framework used to implement application-level security.

To be more explicit, let's discuss using some practical cases. We'll consider a situation in which we deploy a system, as in figure 1.4. This situation is common for a system designed using a microservices architecture, especially if you deploy it in multiple availability zones in the cloud.

With such architectures, we could encounter various vulnerabilities, so you should exercise attention. As mentioned earlier, security is a cross-cutting concern which we design on multiple layers. It's a best practice when addressing the security concerns of one of the layers to assume as much as possible that the above layer doesn't exist. Think about the analogy with the castle in figure 1.2. If you manage the "layer" with the 30 soldiers, you want to prepare them to be as strong as possible. And you do this even knowing that before reaching them, one would need to cross the "bridge of fire".

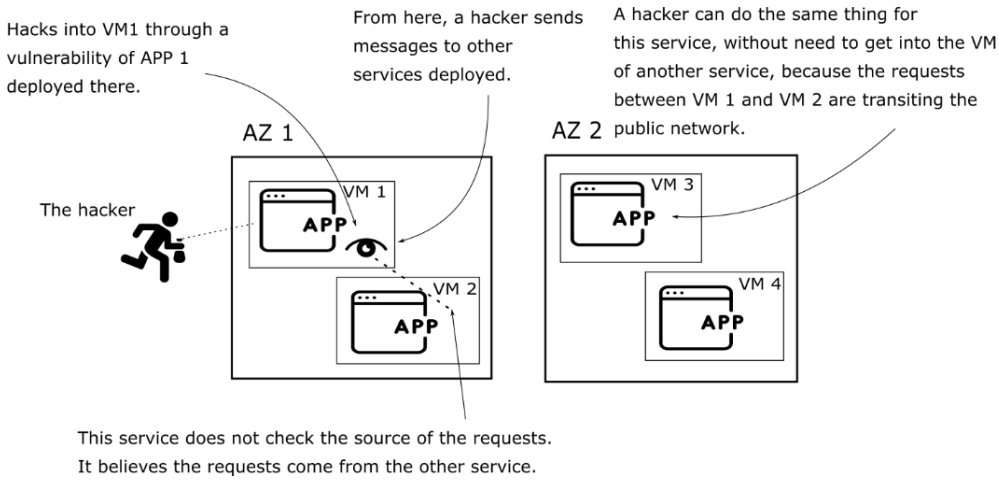

Figure 1.4 If a malicious user manages to get access to the VM, and there's no applied application-level security, a hacker will gain control of the other applications in the system. If the communication is done between two different availability zones (AZ), a malicious individual will find it easier to intercept the messages. This vulnerability allows them to steal data or to impersonate users.

With this in mind, let's consider that an individual driven by bad intentions could log into the virtual machine that's hosting the first application. Let's also assume that no check is done by the second application on the requests sent by the first application. Our attacker could exploit this vulnerability and control the second application by impersonating the first one.

As well, consider that we deploy the two services to two different locations. Then the attacker doesn't need to log in to one of the virtual machines as they can directly act in the middle of the communications between the two applications.

NOTE An *availability zone* in terms of cloud deployment is a separate data center. This data center is situated far enough geographically (and has other dependencies) than other data centers of the same region. This way, it's considered that if one availability zone is failing, the probability that others are failing too is minimal. In terms of security, an important aspect is that traffic between two different data centers generally goes across the public network.

Monolithic and microservices

The discussion on monolithic and microservices architectural styles is a whole different tome. I refer to them from multiple places in this book, so you should at least be aware of the terminology. For an excellent discussion of the two architectural styles, I recommend that you read Chris Richardson's *Microservices Patterns* (Manning, 2018).

(continued)

By *monolithic architecture*, we refer to an application in which we implement all the responsibilities in the same executable artifact. You can see this as one application that fulfills all the use cases. The responsibilities can sometimes be implemented within different modules to help the application be more comfortable to maintain. But the logic of one can't be separated from the logic of others at run time. Generally, monolithically architectures offer less flexibility for scaling.

A *microservice system* has the responsibilities implemented within different executable artifacts. You can see the system as being formed of multiple applications that execute at the same time and communicate between them when needed via the network. While this offers more flexibility for scaling, it introduces other difficulties. We can enumerate here latencies, security concerns, network reliability, distributed persistence, and deployment management.

I referred earlier to authentication and authorization. And indeed, these are most often present in most of the applications. Through authentication, an application identifies a user (a person or another application) to decide afterward what they should be allowed to do—authorization. I'll talk in detail on authentication and authorization, starting with chapter 3 and throughout the book.

In an application, you'll often find the need to implement authorization in different scenarios. Consider another situation: most applications have restrictions regarding who should the user be to access certain functionality. Implementing this implies first the need to identify who creates an access request for a specific feature—authentication. As well, we need to know their privileges to allow them to use that part of the system. As the system becomes more complex, you'll find different situations that require a specific implementation related to authentication and authorization.

For example, what if you'd like to authorize a particular component of the system against a subset of data or operations on behalf of the user? Let's say the printer needs access to read the documents of a user. Should you share the credentials of the user with the printer? But that allows the printer more rights than it needs! And it also exposes the credentials of the user. Is there a proper way to do this without impersonating the user? These are important questions and the kind of questions you encounter when developing applications. Questions that we don't only want to answer, but for which we'll also see applications with Spring Security in this book.

Depending on the chosen architecture for the system, you find authentication and authorization at the level of the entire system, as well as for any of the components. And as we'll see further along in this book, with Spring Security, you'll sometimes prefer to use authorization even for different tiers of the same component. In chapter 13, we'll discuss more on global methods security, which refers to this aspect. The design gets even more complicated because you can have a predefined set of roles and authorities. Moreover, you could be in a situation where you create the roles dynamically.

I also want to bring to your attention the data storage. Data at rest adds to the responsibility of the application. Sometimes a part of the data shouldn't be stored in cleartext. In this case, the application would either choose to keep the data encrypted with a private key or hashed. Secrets, similar to credentials and private keys, can also be considered data at rest. They should be carefully stored, usually in a secrets vault.

> **NOTE** We classify data as "at rest" or "in transition." In this context, data at rest refers to data in computer storage or, in other words, persisted data. Data in transition applies to all the data that's exchanged from one point to another. Different security measures should, therefore, be enforced, depending on the type of data.

Finally, an executing application must manage its internal memory as well. It may sound strange, but data stored in the heap of the application can also present vulnerabilities. Sometimes the class design allows the app to store for a long time, sensitive data such as credentials or private keys. In these cases, someone who has the privilege to make a heap dump could find these details and then use them maliciously.

With a short description of these cases, I hope I've managed to provide you with an overview of what we mean by application security, as well as the complexity of this subject. Software security is a tangled subject. One who's willing to become an expert in this field would, for sure, need to understand (as well as apply) and test solutions for all the layers that collaborate within a system. However, in this book, we focus on presenting all the details for what you specifically need to understand in terms of Spring Security. You'll find out from the previous description where this framework applies and where it doesn't, how it helps, and why you should use it. We'll do this with practical examples that you could adapt to your use cases.

1.3 Why is security important?

The best way to start thinking about why security is important is from your point of view as a user. Like anyone else, you use applications, and they have access to your data. They can change your data, use it, or expose it. Think about all the apps you use, from your email to your online banking service accounts. How would you evaluate the sensitivity of the data that's managed by all these systems? How about the actions that you can perform using these systems? As well as the data, some actions are more important than others. You don't care much about several of them, while others are significant. Maybe for you it's not that important if one could somehow read your emails. But I bet you'd care if someone could empty your bank accounts.

Once you've thought about security from your point of view, try to see a more objective picture. The same data or actions might have another degree of sensitivity to other people. Certain people might care much more than you if their email is accessed. The application should protect everything to the desired degree of access. Any leak that would allow the use of data, functionalities, as well as the application to affect other systems, is considered a vulnerability, and you should solve it.

Not respecting security comes with a price that I'm sure you aren't willing to pay. In general, it's about money. But the cost can differ, and there are multiple ways through which you could lose profitability. It isn't only about directly losing money from a bank account or using a service without paying for it. These are indeed more direct ways that imply costs. The image of a brand or company is also valuable, and losing a good image can be expensive: sometimes even more costly than the direct expenses resulted from the exploit of a vulnerability in the system! The trust that the users have in your application is one of its most valuable assets, and it can make the difference between success or failure.

Here are a few fictitious examples. Think about how would you see them as a user? How can these affect the organization responsible for the software?

1. A back-office application should manage the internal data of an organization but, somehow, information leaks out.
2. Users of a ride-sharing application observe that money is debited from their accounts on behalf of trips that aren't theirs.
3. After an update, users of a mobile banking application are presented with transactions that belong to other users.

In the first situation, the organization using the software, as well as its employees, could be affected. In certain instances, the company could be liable, and a significant amount of money could be lost. In this situation, the users don't have the choice to change the application, but the organization could decide to change the provider of the software system.

In the second case, the users will probably choose to change the service provider. The image would be dramatically affected by the company developing the application. The cost lost in terms of money, in this case, is much less than the cost in terms of image. Even if the payment is returned to the affected users, the application will lose users, which will affect profitability and can even lead to bankruptcy.

The third case could have dramatic consequences on the bank in terms of the trust as well as legal repercussions.

In most of these scenarios, investing in security is safer than what happens if someone exploits a vulnerability in your system. For all the examples, only a small weakness could cause each outcome. For the first example, it could be a broken authentication or a cross-site request forgery (CSRF). For the second or third, it could be a lack of method access control. And for all of them, it could be a combination of vulnerabilities.

From here, we can go even further and discuss the security in defense-related systems. If money is important, add human lives to the cost. Can you even imagine what could be the result if a health system was affected? What about systems that control nuclear power? You can reduce the risk by investing early in the security of the application and by allocating enough time for security professionals to develop and test the security mechanism.

NOTE The lessons learned from those who failed before you are that the cost of an attack is usually higher than the investment of avoiding the vulnerability.

In the rest of this book, you'll see examples of ways to apply Spring Security to avoid situations similar to the ones presented. I guess there can never be enough written words on how important security is. When you have to make a compromise on the security of your system, try to estimate your risks correctly.

1.4 Common security vulnerabilities in web applications

Before discussing how to apply security in applications, you should first know what you're protecting the application from. To do something malicious, an attacker identifies and exploits vulnerabilities of your application. We often describe vulnerability as a weakness that could allow the execution of actions that are unwanted and usually done with malicious intentions.

An excellent start to understanding vulnerabilities is being aware of the Open Web Application Security Project, also known as OWASP (https://www.owasp.org). At OWASP, you'll also find descriptions of the most common vulnerabilities that you should avoid in your applications. Let's take a few minutes and discuss those theoretically before diving into the next chapters, where you'll start to apply concepts from Spring Security. Among the common vulnerabilities that you should be aware of, you'll find:

- Broken authentication
- Session fixation
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Injections
- Sensitive data exposure
- Lack of method access control
- Using dependencies with known vulnerabilities

These items are related to application-level security, and most of them are also directly related to using Spring Security. We'll discuss their relationship with Spring Security and how to protect your application from them in detail in this book, but first an overview.

1.4.1 Vulnerabilities in authentication and authorization

In this book, we discuss authentication in depth. *Authentication* represents the process in which an application identifies someone trying to use it. When someone or something uses the application, we want to obtain their identity so that further access is granted or not. Cases exist in which the access is anonymous, but in most situations, data and actions can be used only by identified requests. Once we have the identity of the user, we can process the authorization.

Authorization is the process of establishing if an authenticated caller has the privileges to use specific functionality and data. For example, in a mobile banking application, most of the authenticated users can transfer money, but only from their account.

We can say that we have a broken authorization if a bad-intentioned individual could somehow gain access to functionality or data which doesn't belong to them. Frameworks such as Spring Security help in making this vulnerability less possible, but if not used correctly, there's still a chance that this might happen. For example, you could use Spring Security to define the access to specific endpoints for an authenticated individual with a particular role. If there's no restriction at the data level, one might find a way to use data that belongs to another user.

Take a look at figure 1.5. An authenticated user can access the `/products/{name}` endpoint. From the browser, this endpoint is called to retrieve and display the user's products from a database. But if the application only checks that the user is authenticated, this allows one user to get the data of another. This situation is one of the examples that should be taken into consideration from the beginning of the design of the application so that you can avoid them.

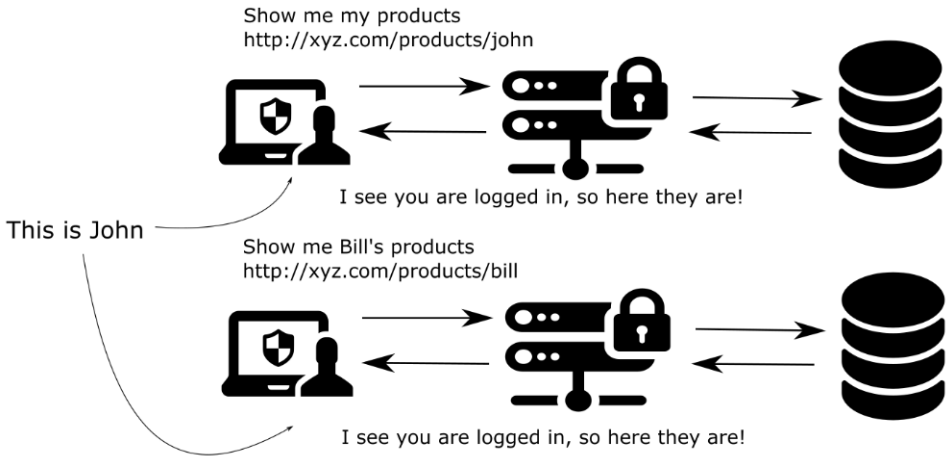

Figure 1.5 A user who's logged in can see their products. But if the application server only checks if the user is logged in, then the user could call the same endpoint to retrieve the products of another user. This way, John could see data that belongs to Bill. The issue that causes the problem is that the application doesn't authenticate the user for data retrieval as well.

Throughout the book, we'll refer to vulnerabilities. We'll discuss them, starting with the basic configuration of the authentication and authorization, in chapter 3. Then we'll see how vulnerabilities relate to the integration of Spring Security and Spring Data and how to design the application to avoid those with OAuth2.

We can classify a weak authentication mechanism also as a broken authentication. In specific situations, there's a need for more than one action to make sure a particular caller is who they claim to be. In this case, a multi-factor authentication (also known as MFA) could be an excellent choice to strengthen the authentication mechanism for the application. In this book, we'll discuss more details about this with examples of MFA done with Spring Security in chapter 9.

1.4.2 What's session fixation?

Session fixation vulnerability is a more specific, high severity weakness of a web application. If present, it could permit an attacker to impersonate a valid user by the reuse of a previously generated session ID. This vulnerability could appear if, during the authentication process, the web application doesn't assign a new session ID, and this could make possible the reuse of existing session IDs. Exploiting this vulnerability consists of obtaining a valid session ID and making the intended victim's browser use it.

Depending on how you implement the web application, you have various ways an individual can use this vulnerability. For example, if the application provides the session ID in the URL, then the victim could be tricked into clicking on a malicious link. If the application uses a hidden attribute, then the attacker can fool the victim into using a foreign form and post the action to the server. If the application stores the value of the session in a cookie, then a script could be injected, and the attacker could force the victim's browser to execute it.

1.4.3 What's cross-site scripting (XSS)?

Cross-site scripting, also referred to as XSS, allows the injection of client-side scripts into web services exposed by the server, thereby permitting other users to run them. Before being used, or even stored, the request should be properly "sanitized" to avoid undesired executions of foreign scripts. The potential impact could relate to account impersonation (for example, combined with session fixation) or to participation in distributed attacks like DDoS.

Let's take an example. A user posts a message or a comment in a web application. After posting the message, the site displays it so that everybody visiting the page can see it. Hundreds could visit this page daily, depending on how popular the site. For the sake of our example, we'll consider it a known site, and a significant number of individuals visit its pages. What if this user posts a script that, when found on a web page, the browser executes (figures 1.6 and 1.7).

1.4.4 What's cross-site request forgery (CSRF)?

Cross-site request forgery (CSRF) vulnerabilities are also common in web applications. CSRF attacks assume a URL that calls an action on a specific server can be extracted and reused from outside of the application (figure 1.8). If the server trusts the execution without doing any check on the origin of the request, one could execute it from any other place. Through CSRF, an attacker could make a user execute undesired actions on a server by hiding the actions. Usually, with this vulnerability, the attacker targets actions that change data in the system.

One of the ways of mitigating this vulnerability is to use tokens to identify the request or use Cross-Origin Resource Sharing (CORS) limitations. With this method, you validate the origin of the request. We look closer at how to deal with CSRF and CORS with Spring Security in chapter 8.

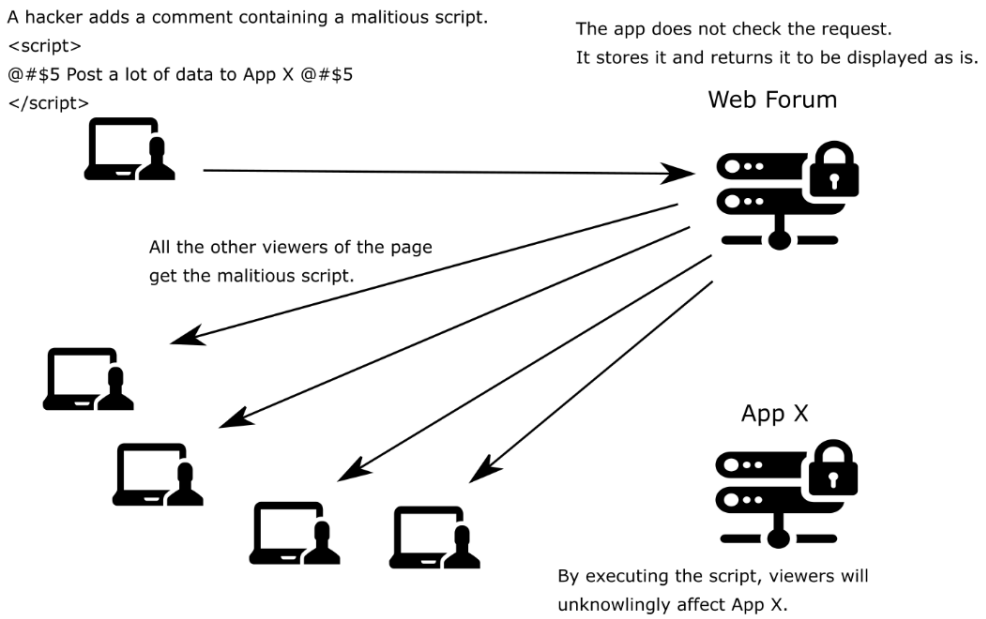

Figure 1.6 **A user posts a comment containing a script on a web forum. The user has defined the script such that it will make requests that try to post or get massive amounts of data from another application (App X)., which represents the victim of the attack. If the Web Forum app allows cross-site scripting (XSS), all the users who display the page with the malicious comment will receive it as it is.**

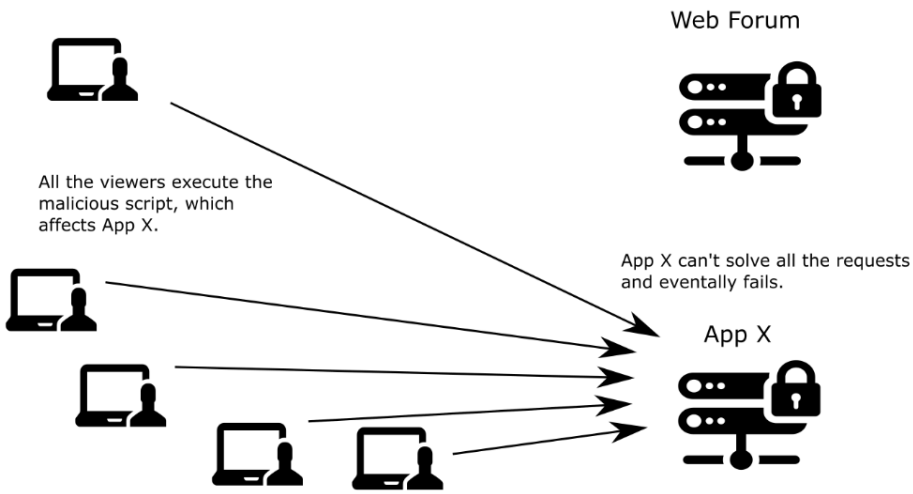

Figure 1.7 **The users access a page that displays a malicious script. Their browsers execute the script and then tries to post or get substantial amounts of data from App X.**

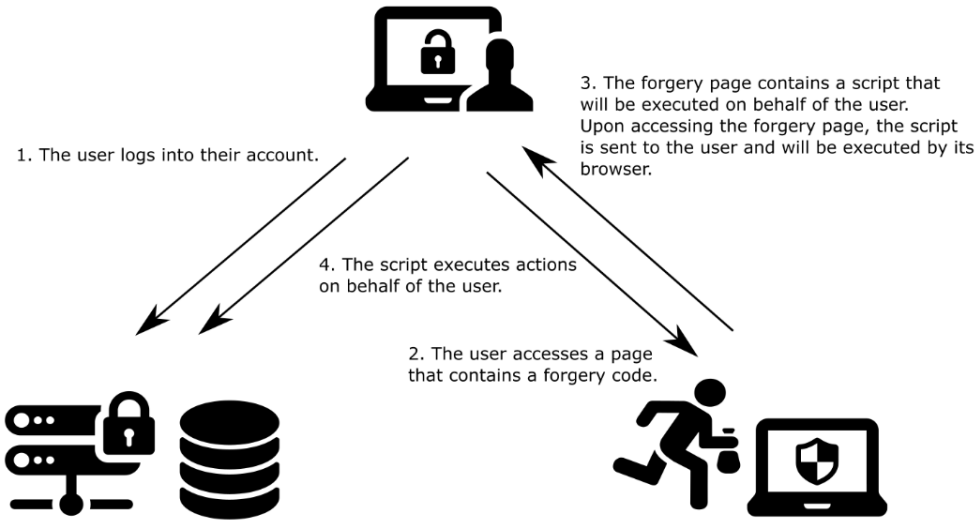

Figure 1.8 Steps of a cross-site request forgery (CSRF). After logging into their account, the user accesses a page that contains forgery code. The malicious code could then execute actions on behalf of the unsuspecting user.

1.4.5 Understanding injection vulnerabilities in web applications

Injection attacks on systems are very common. In an injection attack, the attacker, employing a vulnerability, introduces specific data into the system. The purpose is to harm the system, change data in an unwanted way, or retrieve data that's not meant to be accessed by them.

Many types of injection attacks exist. Even the XSS that we mention in section 1.4.3 can be considered an injection vulnerability. In the end, injection attacks inject a client-side script with the means of harming the system somehow. Other examples could be SQL injection, XPath injection, OS command injection, LDAP injection, and the list continues.

Injection types of vulnerabilities are important, and the results of exploiting them could be change, deletion, or access to data in the systems being compromised. For example, if your application is somehow vulnerable to LDAP injection, an attacker could try to benefit from bypassing the authentication and, from there, control essential parts of the system. The same could happen for XPath injection or OS command injection.

One of the oldest, and maybe the most known type of injection vulnerability, is SQL injection. If your application has an SQL injection vulnerability, an attacker could try to change or run different SQL queries to alter, delete, or extract data from your system. In the most advanced SQL injection attacks, an individual can run OS commands on the system, and this would lead to a full system compromise.

1.4.6 Dealing with the exposure of sensitive data

Even if, in terms of complexity, the disclosure of confidential data seems to be the easiest to understand and the least complex of the vulnerabilities, it remains one of the most common mistakes. Maybe this happens because the majority of tutorials and examples found online as well as books illustrating different concepts define the credentials directly in the configuration files for simplicity reasons. In the case of a hypothetical example that eventually focuses on something else, this makes sense. Developers learn continuously and see this happening in all the examples without also seeing mentions of what's wrong with this approach. Because of this, they might mistakenly think that this is a good practice.

How is this related to Spring Security? Well, we'll deal with credentials and private keys in the examples in this book. We might use secrets in configuration files, but we'll place a note for these cases to remind us that sensitive data should be stored in vaults. Naturally, for a developed system, the developers aren't allowed to see the values for these sensitive keys in all of the environments. Usually, at least for production, only a small group of people should be allowed to access the private data.

By setting such values in the configuration files, such as the application.properties or application.yml files in a Spring Boot project, you make the values accessible to anyone who can see the source code. Moreover, you might also find yourself storing all the history of these value changes in your version management system for source code.

Also related to the exposure of sensitive data is the information in logs written by your application to the console or stored in databases such as Splunk or Elasticsearch. I have often seen logs that disclose sensitive data forgotten by the developers.

> **NOTE** Never log something that isn't public information. By public, I mean something that can be seen or accessed by anyone. Things such as private keys or certificates aren't public and shouldn't be logged together with your error, warnings, or information messages.

Next time you log something from your application, make sure it doesn't look like one of the messages below:

```
[error] The signature of the request is not correct. The correct key to be
     used should have been X.

[warning] Login failed for username: X and password Y. User with username X
     has password Z.

[info] A login was performed with success by the user X with password Y.
```

Be careful of what your server returns to the client, especially, but not limited to the cases when the application encounters exceptions. Often, by lack of time or experience, developers forget to implement all the cases. This way, and usually happening after a wrong request, the application returns too many details, which exposes the

implementations. This application behavior is also a vulnerability through data exposure. If your application encountered a `NullPointerException` because the request was wrong (part of it was missing, for example), then the exception shouldn't appear in the body of the response. At the same time, the HTTP status should be 400 rather than 500. HTTP status codes of type 4XX are designed to represent problems on the client-side. A wrong request is, in the end, an issue of the client, so the application should represent it accordingly. HTTP status codes of type 5XX are designed to inform that there was a problem on the server. Do you see something wrong in the response presented by the next snippet?

```
{
    "status": 500,
    "error": "Internal Server Error",
    "message": "Connection not found for IP Address 10.2.5.8/8080",
    "path": "/product/add"
}
```

The message of the exception seems to be disclosing an IP address. An attacker could use this address to understand the network configuration and eventually find a way to control the virtual machines in your infrastructure. With only this piece of data, however, one couldn't do any harm. But collecting different disclosed pieces of information and putting them together could provide everything that's needed to adversely affect a system. Having exception stacks in the response is not a good choice either; for example:

```
at
 java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolEx
 ecutor.java:1128) ~[na:na]
at
 java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolE
 xecutor.java:628) ~[na:na]
at
 org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThrea
 d.java:61) ~[tomcat-embed-core-9.0.26.jar:9.0.26]
at java.base/java.lang.Thread.run(Thread.java:830) ~[na:na]
```

Not only that, this approach discloses the application's internal structure. From the stack of an exception, you could see the naming notations as well as objects used for specific actions and the relationships between them. But even worse than that, logs sometimes can disclose versions of dependencies that your application uses. (Did you spot that Tomcat-core version in the above exception stack?)

We should avoid using vulnerable dependencies. However, if we find ourselves using a vulnerable dependency by mistake, at least we don't want to point this mistake out explicitly. Even if the dependency isn't known as a vulnerable one, this could be because nobody has found the vulnerability yet. Exposures as the previous snippet could motivate an attacker to find vulnerabilities in that specific version because they know now that's what your system uses. It's inviting them to harm your system. And an attacker could use even the smallest detail against a system.

```
Response A:
{
    "status": 401,
    "error": "Unauthorized",
    "message": "Username is not correct",
    "path": "/login "
}
Response B:
{
    "status": 401,
    "error": " Unauthorized",
    "message": "Password is not correct",
    "path": "/login "
}
```

In this example, the responses A and B are different results of calling the same authentication endpoint. They don't seem to expose any information related to the class design or system infrastructure, but they hide another problem. If the messages disclose context information, then they can as well hide vulnerabilities. The different messages based on different inputs provided to the endpoint can be used to understand the context of execution. In this case, they could be used to know when a username is correct, and only the password is wrong. And this can make the system more liable to a brute force attack. The response provided back to the client shouldn't help in identifying a possible guess of a specific input. In this case, it should have better provided in both situations the same message:

```
{
    "status": 401,
    "error": " Unauthorized",
    "message": "Username or password is not correct",
    "path": "/login "
}
```

It could look small, but if not taken and in the right context, exposing sensitive data could become an excellent tool to be used against the system.

1.4.7 What's the lack of method access control?

Even at the application level, you don't apply authorization to only one of the tiers. Sometimes it's a must to make sure that a particular use case can't be called at all (for example, if the privileges of the currently authenticated user don't allow it).

Imagine a situation where the authorization is done only at the endpoint level (assuming that you can access the method through a REST endpoint). A developer might be tempted to apply authorization rules only in the controller layer, as presented in figure 1.9.

While the case presented in figure 1.9 works correctly, applying the authorization rules only at the controller layer might leave room for error. In this case, a future implementation could expose that use case without testing or without testing all the authorization requirements. In figure 1.10, you can see what could happen if a developer adds another functionality that depends on the same repository.

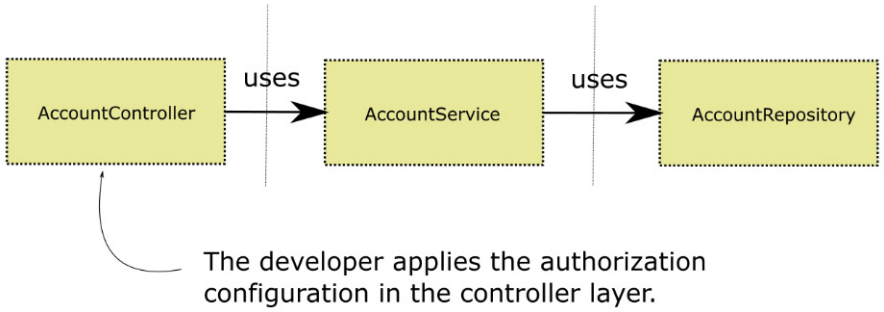

Figure 1.9 **A developer applies the authorization rules at the controller layer. But the repository doesn't know and doesn't restrict the retrieval of data anyhow. If a service asked for accounts that don't belong to the currently authenticated user, the repository would return them.**

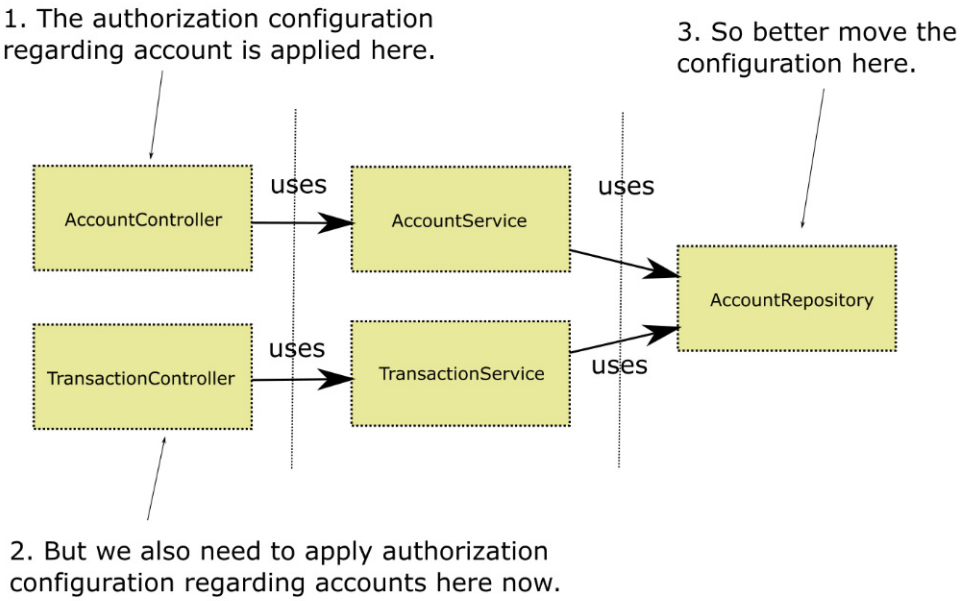

Figure 1.10 **The newly added TransactionController makes use of the AccountRepository in its dependency chain. The developer must reapply the authorization rules in this controller as well. But it would be much better if the repository itself made sure that data that doesn't belong to the authenticated user isn't exposed.**

These situations might appear, and you might need to treat them at any layer in your application, not only with the repository. We'll discuss more things related to this subject in chapter 13. There, you'll see how we can apply restrictions per application tier when this is needed, as well as the cases when we should avoid doing this.

1.4.8 Using dependencies with known vulnerabilities

Although not necessarily related directly to Spring Security, but still an important aspect of the application-level security, we come to paying attention to the dependencies used. Sometimes it's not the application you develop that has vulnerabilities, but the dependencies like libraries or frameworks that you use to build the functionality. You should always be attentive to the dependencies you use and eliminate any version that's known to contain a vulnerability.

Fortunately, we have multiple possibilities for static analyses, quickly done by adding a plugin to your Maven or Gradle configuration. The majority of the applications today are developed based on open source. Even Spring Security is an open-source framework. This development methodology is great, and it allows for fast evolution, but this rush could also make us more error-prone.

This rush in development is why, when developing any piece of software, we have to take all the needed measures to make sure that we avoid the use of any dependency that was proven to have known vulnerabilities. If we discover we've used such a dependency, then we not only have to correct this fast, we also have to investigate if the vulnerability was already exploited and take the needed measures.

1.5 Security applied in various architectures

In this section, we'll discuss applying security practices depending on the design of your system. It's important to understand that different software architectures imply different possible leaks and vulnerabilities. I want to make you aware of this first chapter about this philosophy to which I'll refer throughout the book. Architecture strongly influences the choices in configuring Spring Security for your applications, so do the functional and non-functional requirements. Even when you think of a real-life situation, to protect something, depending on what you want to protect, you'll use a metal door, bulletproof glass, or a barrier. You couldn't use only a metal door in all the situations. If what you protect is an expensive painting in a museum, you still want people to be able to see it. You don't want them to be able to touch it, damage it, or even take it with them. In this case, the functional requirements are the ones affecting the solution we take for secure systems.

It could be that you need to make a good compromise with other quality attributes, such as performance. It's like using your heavy metal door instead of a lightweight barrier at the parking entrance. You could do that, and for sure, the metal door would be more secure, but it takes much more time to open and close it. The time and cost of opening and closing the heavy door aren't worth it. We're assuming that this isn't special parking for expensive cars.

Because the security approach is different depending on the solution we implement, the configuration in Spring Security is also different. In this section, we discuss several examples based on different architectural styles as well as other requirements, how the security approach is affected, and what we should take into consideration.

These aspects are linked to all the configurations that we'll work on in the following chapters with Spring Security.

In this section, I present several of the practical scenarios you might have to deal with and with which we'll work over in the rest of the book. For a more detailed discussion on these aspects, I recommend you also read the *Microservices Security in Action* by Prabath Siriwardena and Nuwan Dias (Manning 2019).

1.5.1 Designing a one-piece web application

Let's start with the case where you develop a component of the system that represents a web application. In this application, there's no direct separation in development between the backend and the frontend. The way we usually see these kinds of applications is through the general servlet flow: the application receives an HTTP request and sends back an HTTP response to a client. Sometimes we might have a server-side session for each client to store specific details over more HTTP requests. In the examples provided in the book, we use Spring MVC (figure 1.11).

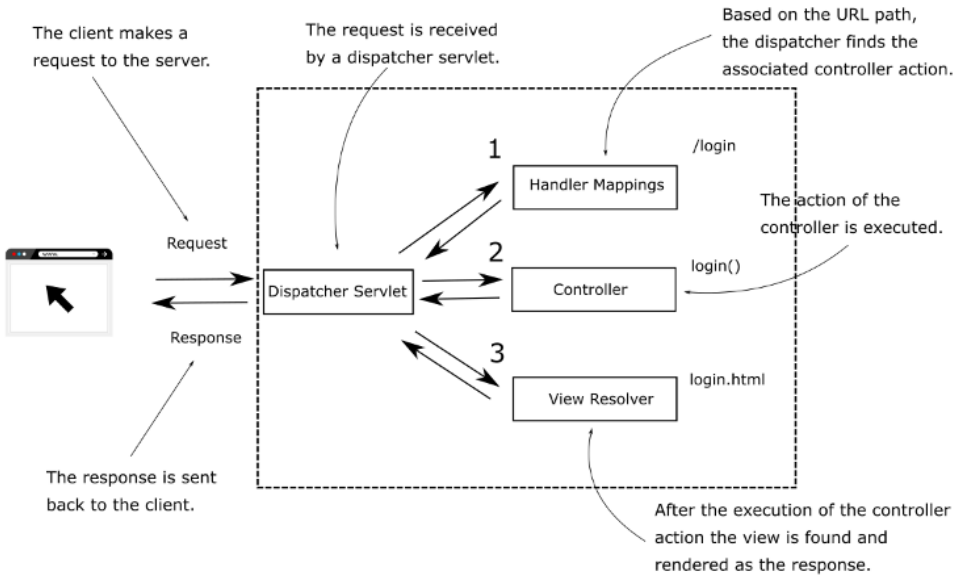

Figure 1.11 A minimal representation of the Spring MVC flow. The Dispatcher Servlet finds the mapping of the requested path to the controller method (1), executes the controller method (2), and obtains the rendered view (3). The HTTP response is then delivered back to the requester, whereby the browser interprets and displays the response.

You'll find a great discussion about developing web applications and REST services with Spring in Craig Walls's *Spring In Action*, Fifth Edition (Manning, 2018):

https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-2/
https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-6/

As long as you have a session, you have to take into consideration the session fixation vulnerability as well as the CSRF possibilities previously mentioned. You must also consider what you store in the HTTP session itself.

Server-side sessions are quasi-persistent. They're stateful pieces of data, so their lifetime is longer. The longer they stay in the memory, the more it's statistically probable that they'll be accessed. For example, a person who has access to the heap dump could also read the information in the application's internal memory. And don't think that the heap dump is challenging to obtain. Especially when developing your applications with Spring Boot, you might find that the actuator is also part of your application.

The Spring Boot actuator is a great tool, and it will also be part of several of our examples. Depending on how you configure it, the Spring Boot actuator could return a heap dump with only an endpoint call. You don't necessarily need root access to the VM to get your dump. We'll discuss in more detail the actuator's configuration in terms of security, as well as the link between the actuator and Spring Security, in the last chapter of this book.

Going back to the vulnerabilities. With CSRF, in this case, the easiest way to mitigate the vulnerability would be to use anti-CSRF tokens. Fortunately, with Spring Security, this capability is available out of the box. CSRF protection, as well as validation of the origin CORS, is enabled by default. You'll have to disable it if you don't want it explicitly. For authentication and authorization, you could choose to use the implicit login form configuration from Spring Security. With this, you'd benefit from only needing to override the look-and-feel of the login and logout and from the default integration with the authentication and authorization configuration. You'd also benefit from the mitigation of the session fixation vulnerability.

If you've authentication and authorization, it also means that you should have users with valid credentials. Depending on your choice, you could have your application managing the credentials for the users, or you could choose to benefit from another system to do this (for example, you might want to let the user log in with their Facebook, GitHub, or LinkedIn credentials). In any of these cases, Spring Security helps you with a relatively easy way of configuring user management. You can choose to store user information in a database, use a web-service, or connect to another platform. The abstractions used in Spring Security's architecture make it decoupled, which allows you to choose any implementation fit for your application.

1.5.2 Designing security for a backend/frontend separation

Nowadays, we more often see in the development of web applications a choice in the segregation of the frontend and the backend (figure 1.12). In these web applications, developers use today a framework like Angular, ReactJS, or Vue.js to develop the frontend. The frontend communicates with the backend through REST endpoints. We'll implement examples to prove applying Spring Security for these architectures, starting with chapter 10.

We typically avoid using server-side sessions; client-side sessions replace them. This kind of system design is also similar to the one used in the case of mobile applications.

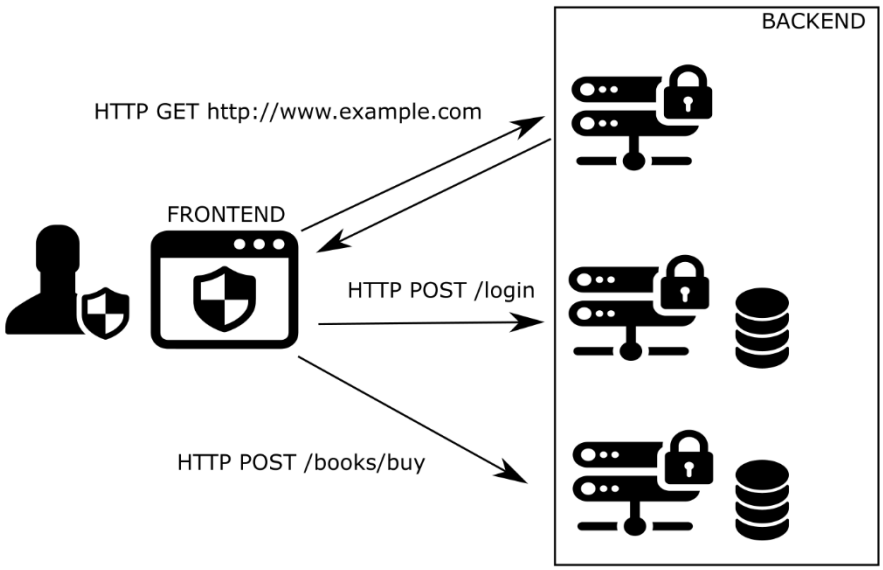

Figure 1.12 The browser executes a frontend application. This application calls REST endpoints exposed by the backend to perform operations requested by the user.

Applications that run on Android or iOS operating systems, which can be native or simple progressive web applications, would call a backend through REST endpoints.

In terms of security, other aspects should be taken into consideration. First, CSRF and CORS configurations are usually more complicated. It's complex because you might want to scale the system horizontally and not necessarily to have the frontend with the backend at the same origin. For mobile applications, we can't even talk about origin. The most straightforward but least desirable approach in a practical solution would be to use HTTP Basic for the endpoint authentication. While this approach is easy to understand and generally used with the first theoretical examples of authentication, it does have leaks that you'd want to avoid. For example, using HTTP Basic implies sending the credentials with each call. As you'll see in chapter 2, credentials aren't encrypted. The browser sends those as a Base64 encoding. Leaving them available on the network in the header of each endpoint call couldn't be considered good practice. Second, assuming that the credentials represent the user that's logged in, you don't want the user to type the credentials for every request. You also don't want to have to store the credentials on the client-side. This practice is again not advisable.

Having the above reasons in mind, you'll see in chapter 10, an alternative that offers a better approach: the OAuth2 flow, but the following section provides an overview for you.

A short reminder of application scalability

Scalability refers to the quality of a software application in which it can serve more or fewer requests while adapting the resources used, without a need to change it or its architecture. Mainly we classify scalability in two types: vertical and horizontal.

When a system is scaled vertically, the resources of the system on which it executes are adapted to the need of the application (for example, when there are more requests, more memory and processing power is added to the system).

We accomplish horizontal scalability by changing the number of instances of the same application that are in execution (for example, if there are more requests, one more instance is started to serve the increased need). If the demand decreases, we can reduce the instances numbers as well. I'm assuming the newly spun-up application instances consume resources offered by additional systems, sometimes even in multiple datacenters.

1.5.3 Understanding the OAuth2 flow

In this section, I give a high-level overview of the OAuth2 flow, and explain the reason for OAuth2 and how it relates to what we've discussed in section 1.5.2. We'll discuss this topic in detail in chapters 10, 11, and 12. We'll also write the components and implement the OAuth2 flow with Spring Security in those chapters. We certainly want to find a solution to avoid resending credentials for each of the requests to the backend and store them on the client-side. The OAuth2 flow offers a better way to implement authentication and authorization in this case.

The OAuth2 flow defines two separate entities: the authorization server and the resource server. The purpose of the authorization server is to authorize the user and provide them with a token that specifies, among other things, a set of privileges that can be used. The part of the backend implementing the functionality is called the resource server. The endpoints that can be called are considered protected resources. Based on the obtained token, after accomplishing the authorization, a call on a resource will be permitted or rejected. Figure 1.13 presents a general picture of the standard OAuth2 authorization flow. Step by step, the following happens:

1. The user accesses a use case in the application (also known as the client). The application needs to call a resource in the backend.
2. To be able to call the resource, the application first has to obtain an access token, so it calls the authorization server to get the token. In the request, it sends the user's credentials or a refresh token in certain cases.
3. If the credentials or the refresh token are correct, the authorization server returns a (new) access token to the client.
4. The access token is used in the header of the request to the resource server when calling the needed resources.

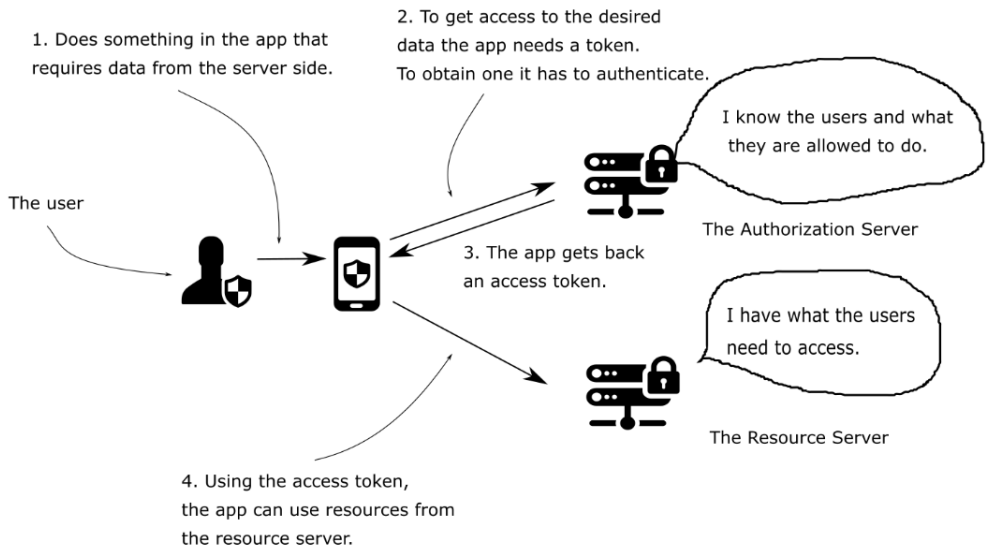

Figure 1.13 The OAuth 2 authorization flow with password grant type. To execute an action requested by the user (1), the application requires an access token from the authorization server (2). The application receives a token (3) and accesses a resource from the resource server with the access token (4).

The token is similar to an access card for a building. As a visitor, you first visit the front desk, where you receive an access card after identifying yourself. The access card can open some of the doors, but not necessarily all. Based on your identity, you can access exactly the doors that you're allowed to and no more. The same happens with an access token. After the authentication, the caller is provided with a token, and based on that, they can access the resources for which they have privileges.

A token has a determined lifetime, usually being short-lived. Depending on the implementation, when a token expires, a new authorization should be made either with a refresh token or with user credentials. If needed, the token can be disqualified by the server earlier than its expiration time. The following lists several of the advantages of this flow:

- The client doesn't have to store the user's credentials. The access token and, eventually, the refresh token are the only access details needed to be saved.
- The application doesn't expose the user's credentials that are often on the network.
- If someone intercepts a token, you can disqualify the token without needing to invalidate the user's credentials.
- A token can be used by a third entity to access resources on the user's behalf, without having to impersonate the user. An attacker could steal the token in this case. But, because the token usually has a limited lifespan, the timeframe in which one can use this vulnerability is limited.

NOTE To make it simple and only give you an overview I've described you with the OAuth2 flow which is called the password grant type. OAuth2 defines multiple grant types and, as you'll see in chapters 10, 11, and 12, it's not always that the client application has the credentials. If we were using the authorization code grant, the application would have redirected the authentication in the browser directly to a login implemented by the authorization server. But more on this later in the book.

Not everything is perfect even with the OAuth2 flow, and you need to adapt it to the application design. One of the questions could be: which is the best way to manage the tokens? In the examples we'll work on in chapters 10, 11, and 12, we cover multiple possibilities that include:

- Persisting the tokens in memory
- Using a database to persist the tokens
- Using cryptographic signatures with JSON Web Tokens (JWT)

1.5.4 Using API keys, cryptographic signatures, and IP whitelisting to secure requests

In certain cases, you don't need a username and a password to authenticate and authorize a caller, but you still want to make sure that nobody altered the exchanged messages. You might need this approach when the requests are made between two backend components. Sometimes you'd like to make sure that the messages between them are validated somehow (for example, if you deploy your backend as a group of services or you use another backend external to your system). In this direction, a few practices include:

- Using static keys in request and response headers
- Signing the requests and response with cryptographic signatures
- Applying IP addresses whitelisting

The use of static keys is the weakest approach. In the headers of the request and the response, we use a key. The request and responses aren't accepted if the header value is incorrect. This assumes that we often exchange the value of the key in the network; if the traffic goes outside the data center, it would be easy to intercept. Someone who gets the value of the key could replay the call on the endpoint. When we use this approach, it's usually done together with IP address whitelisting.

A better approach to test the authenticity of the communication is the use of cryptographic signatures (figure 1.14). With this approach, a key is used to sign the request and the response. You don't need to send the key on the wire, which is an advantage over static authorization values. The parties can use their key to validate the signature. The implementation can be done using two asymmetric key pairs. This approach assumes that we never exchange the private key. There's also the simpler version in

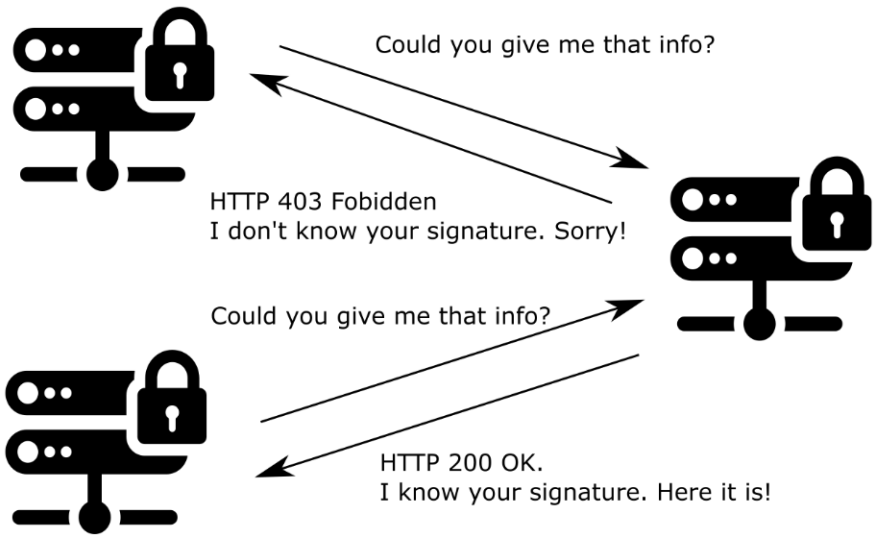

Figure 1.14 **To make a successful call to another backend, the request should have the correct signature or shared key.**

which we use a symmetric key, which requires a first-time exchange for configuration. The disadvantage is that the computation of a signature consumes more resources.

If you know an address or range of addresses from where the request should come from, then together with one of the solutions mentioned previously, IP address whitelisting can be applied. This method implies that the application will reject the requests if coming from other IP addresses than the ones that are configured to be whitelisted. However, most of the cases, IP whitelisting isn't done at the application level, but much earlier, by the networking layer.

1.6 What will you learn in this book?

This book offers a practical approach to learning Spring Security. Throughout the rest of the book, we'll deep dive into Spring Security step-by-step, proving concepts with simple to more complex examples. To get the most out of this book, you should be comfortable with Java programming, as well as with the basics of the Spring framework. If you haven't used the Spring framework, you should read *Spring Framework In Action*, Fifth Ed., by Craig Walls (Manning 2018), and another great resource is also *Spring Boot In Action* by Craig Walls (Manning 2015).

In the book you're reading now, you'll learn:

- The architecture and basic components of Spring Security and how to use them to secure your application
- Authentication and authorization with Spring Security including the OAuth2 and OpenID Connect flows and how they apply to a production-ready application

- How to implement security with Spring Security on different layers of your application
- Different configuration styles and the best practices for using those in your project
- What you should know about deploying an application secured with Spring Security in a containerized environment

In this book, to make the learning process smooth for each described concept, we'll work on multiple simple examples. At the end of each significant subject, we'll review the essential concepts you've learned with a more complex application. You'll find these sections in the book with the name "Hands-On."

When we finish, you'll know how to apply Spring Security for the most practical scenarios and understand where to use it and its best practices. I also strongly recommend that you work on all the examples that accompany the explanations.

1.7 Summary

- Spring Security is the leading choice for securing Spring applications. It offers a significant number of alternatives that apply to different styles and architectures.
- You should apply security in layers for your system, and for each layer, you should use different practices.
- Spring Security is a project related to application-level security.
- Security is a cross-cutting concern, and you should consider it from the beginning of a software project.
- Usually, the cost of an attack is higher than the investment in avoiding vulnerabilities.
- The Open Web Application Security Project (OWASP) is an excellent place to start, then always refer to that when it comes to vulnerabilities and security concerns.
- Sometimes the smallest mistakes can cause significant harm. For example, exposing sensitive data through logs or error messages is a common way to introduce vulnerabilities in your application.