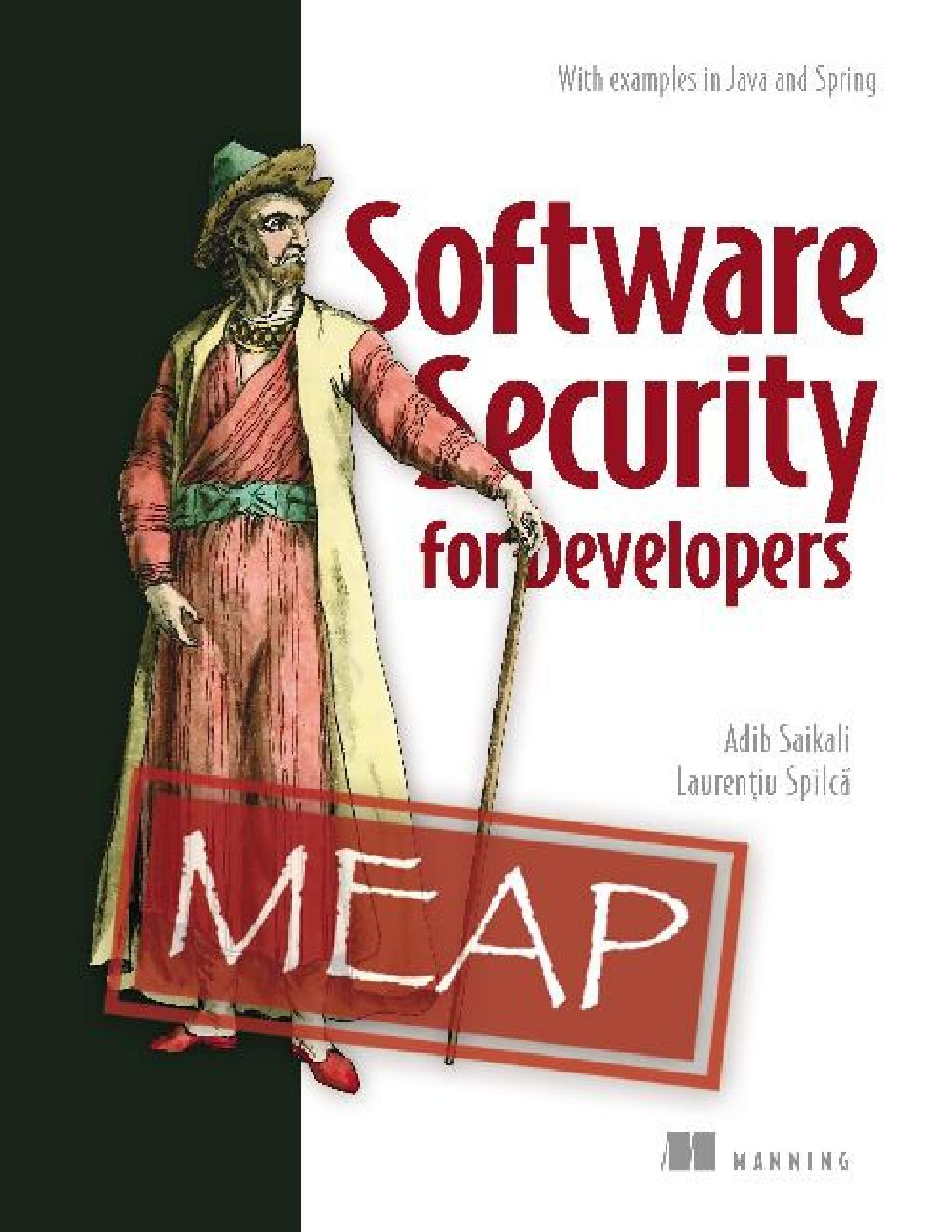


With examples in Java and Spring

A man dressed in a green coat and a straw hat is carrying a large, red book with the letters "MEAP" written on it. He is walking towards the right side of the frame.

Software Security for Developers

Adib Saikali
Laurentiu Spilcă



MANNING

With examples in Java and Spring

Software Security for Developers

Adib Saikali
Laurențiu Spilcă



MANNING

Software Security for Developers

1. [welcome](#)
2. [Part 1: Application Security the Big Picture](#)
3. [1 Making Sense of Application Security](#)
4. [2 Standards for implementing authentication](#)
5. [3 Service-to-service communication](#)
6. [Part 2: Cryptography foundations](#)
7. [4 Message Integrity and Authentication](#)
8. [5 Advanced Encryption Standard](#)
9. [6 Public Key Encryption and Digital Signatures: Unleashing RSA](#)
10. [7 Public Key Encryption and Digital Signatures: Using ECC](#)
11. [Part 3: Securing communication channels](#)
12. [8 Public Key Infrastructure and X.509 Digital Certificates: know w](#)
13. [9 Working with X.509 Certificates: Lifecycle and Self-Signing](#)
14. [10 Transport Layer Security \(TLS\): How the internet is secured](#)
15. [Part 4: Modern Authentication and Identity](#)
16. [11 JSON Object Signing and Encryption \(JOSE\)](#)
17. [12 Single Sign On \(SSO\) using OAuth2 and OpenID Connect](#)
18. [13 Deepening security with OpenID Connect](#)
19. [14 Passwordless login: Using Magic links and OTPs](#)
20. [15 Passwordless login: WebAuthn and hardware authentication](#)
21. [Part 5: Securing service-to-service call chain](#)
22. [16 Implementing service identity](#)
23. [17 Taming authorization: RBAC, ABAC, ReBAC](#)
24. [Appendix A. Installation and Setup](#)

welcome

Thank you for purchasing the MEAP edition of *Software Security for Developers: With examples in Java and Spring*. This book is for developers who want to learn application security in a practical way using sample applications to explore complex security protocols, algorithms, and patterns.

Over the past 20 years I have implemented security on numerous applications, which meant correctly configuring and using a variety of security libraries and protocols. For example, implementing Single Sign On using SAML or OpenID Connect, or encrypting files with AES, or configuring TLS cipher suites on a tomcat server. I frequently got stuck on security related error messages I did not understand, on security APIs that seemed hard to use, so I invested a lot of time and effort to learn security. This meant a lot of time and effort reading a lot of books with a lot of math in them to learn the background required to correctly and easily use security protocols required to build modern applications.

I am writing the book I wish I had when I started learning security as a developer. This book is focused on security use cases you need to implement in applications. By the end of the book, you will know how to:

- Use industry standard cryptography algorithms correctly
- Implement Single Sign On using OpenID Connect
- Get rid of passwords using the Web Authentication Protocol
- Configure and debug mutual TLS connections easily and correctly
- Store and access application secrets in the most popular key management services.
- Securely containerize your application and then Lockdown it down tight on Kubernetes
- Use API gateways and service mesh to secure the service-to-service call chain

I assume you are a developer who can read Java code, but don't have a deep security background. The book provides you with all the required background

and concepts you need. For each concept you will have a working application that you run, put break points on, and study so that you can learn what you need to learn. To make the dry topics of cryptography easier to comprehend I have built sample applications using libraries and standards you are likely to encounter. Please be patient and work your way through the book chapter by chapter, so you don't get lost by skipping ahead and then finding that you are missing some background. If you go through the book chapter by chapter and run the sample apps, everything will make sense.

Please be sure to post any questions, comments, or suggestions you have about the book in the [liveBook discussion forum](#).

—Adib Saikali

In this book

[welcome](#) [Part 1: Application Security the Big Picture](#) [1 Making Sense of Application Security](#) [2 Standards for implementing authentication](#) [3 Service-to-service communication](#) [Part 2: Cryptography foundations](#) [4 Message Integrity and Authentication](#) [5 Advanced Encryption Standard](#) [6 Public Key Encryption and Digital Signatures: Unleashing RSA](#) [7 Public Key Encryption and Digital Signatures: Using ECC](#) [Part 3: Securing communication channels](#) [8 Public Key Infrastructure and X.509](#) [Digital Certificates: know who you are talking to](#) [9 Working with X.509 Certificates: Lifecycle and Self-Signing](#) [10 Transport Layer Security \(TLS\): How the internet is secured](#) [Part 4: Modern Authentication and Identity](#) [11 JSON Object Signing and Encryption \(JOSE\)](#) [12 Single Sign On \(SSO\) using OAuth2 and OpenID Connect](#) [13 Deepening security with OpenID Connect](#) [14 Passwordless login: Using Magic links and OTPs](#) [15 Passwordless login: WebAuthn and hardware authentication](#) [Part 5: Securing service-to-service call chain](#) [16 Implementing service identity](#) [17 Taming authorization: RBAC, ABAC, ReBAC](#) [Appendix A. Installation and Setup](#)

Part 1: Application Security the Big Picture

Computer security is a vast field with many different technologies that must be learned independently then combined correctly in an application.

Application developers and architects typically learn security technologies on the job when they first encounter them while under pressure to deliver product features and bug fixes. Reading blog posts, cutting and pasting configuration settings, and searching stackoverflow.com for help while under pressure to deliver leaves developers feeling like they don't understand security but also don't have the time and resources to properly learn it.

A step-by-step plan that breaks security technologies into easily digestible chunks that a developer or architect can learn quickly and independently on the job is the goal of part 1. The plan starts by building a mental model of cloud native application security. The model allows you to definitely answer the following questions.

- What security technologies do you need to know to implement security on the application you are currently working on?
- What is the correct order to learn security technologies in so that you don't get stuck because you don't understand a dependency of the technology you are learning?
- What level of depth should you aim for when learning a security technology?
- What is the division of roles and responsibilities between application developers, architects, cloud automation engineers, infrastructure providers, and security engineers?

Part 1 will help you grasp the big picture, connect the dots between the security standards and technologies widely used for cloud native applications, enabling you to zoom in on the relevant parts of the book for your needs.

1 Making Sense of Application Security

This chapter covers

- Why learning security is important
- The consequences of poor security practices
- What are the roles in the information security field
- Identifying the security skills you should possess as an application developer

Every week we are treated to a headline about some security vulnerability in a widely used piece of software or a data breach at a mega-corporation affecting millions of users (figure 1.1). My bank replaced my credit card twice in a five-year period due to data breaches at large retailers where I shopped.

Figure 1.1 Headlines showcasing major recent data breaches and security vulnerabilities, emphasizing the widespread impact on millions of users and the persistent threat to digital security.



By Reuters

October 25, 2024 12:01 PM GMT+3 · Updated 22 days ago



AMAZON / TECH / SECURITY

Amazon confirms employee data breach, but says it's limited to contact info



/ Work contact addresses, phone numbers, building locations, a leak that occurred last year.

Reuters World · US Election Business · Markets · Sustainability · Legal · Breakingviews · Technology · M

US reaches \$31.5 million settlement with T-Mobile over data breaches

By David Shepardson

September 30, 2024 11:16 PM GMT+3 · Updated 2 months ago



POLITICO

EXCLUSIVE

Chinese hackers gained access to huge trove of Americans' cell records

Investigators aren't sure how much data Salt Typhoon might have taken, and are still struggling to evict the elite Chinese hacking crew from company networks.

Reuters

World · US Election Business · Markets · Sustainability · Legal · Breakingviews

AT&T to pay \$13 million over 2023 customer data breach

By David Shepardson

September 18, 2024 12:31 AM GMT+3 · Updated 2 months ago



We used to think that security vulnerabilities are primarily a software issue. However, hardware security vulnerabilities have been common in recent years. The Meltdown and Specter vulnerabilities, disclosed in January 2018, allowed attackers to bypass CPU hardware protections and gain unauthorized memory access (<https://meltdownattack.com/>). In a cloud or multi-tenant

environment, Meltdown and Specter make it possible for one cloud tenant to see the memory of another tenant. The hardware walls we depend on to isolate workloads were suddenly full of holes for attackers to sneak through.

The past few years have taught us that every layer of the stack, from hardware all the way to JavaScript in a web browser, can have security vulnerabilities. Security is everyone's collective responsibility to build and run IT systems, from hardware engineers who design the processors in our phones to application developers who build the e-commerce applications that keep our kitchens stocked with food. Regardless of your role in the IT world, security is your responsibility.

Note

Regardless of your role in the IT world security is your responsibility.

1.1 Today's reality – tradeoffs and costs

The average cost of a data breach in 2020 is \$3.92 million (<https://www.csionline.com/article/3434601/what-is-the-cost-of-a-data-breach.html>). Some mega breaches, such as Marriot Hotels leaked 500 million (<https://www.wsj.com/articles/marriott-take-126-million-charge-related-to-data-breach-11565040121>) customer records. Marriott took a \$126 million charge to deal with the data breach. Equifax an American credit reporting agency spent 1.4 billion (<https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473>) dollars on cleanup costs associated with the 2017 data breach of 150 million personal credit histories.

A significant security incident has the potential to be a company ending event. CEOs are quite concerned about the business impact of security breaches, so they appoint a Chief Information Security Officer (CISO) reporting directly to them as a peer of the Chief Information Officer (CIO). Reporting directly to the CEO enables the CISO to make the necessary organizational and technology changes to secure corporate IT systems.

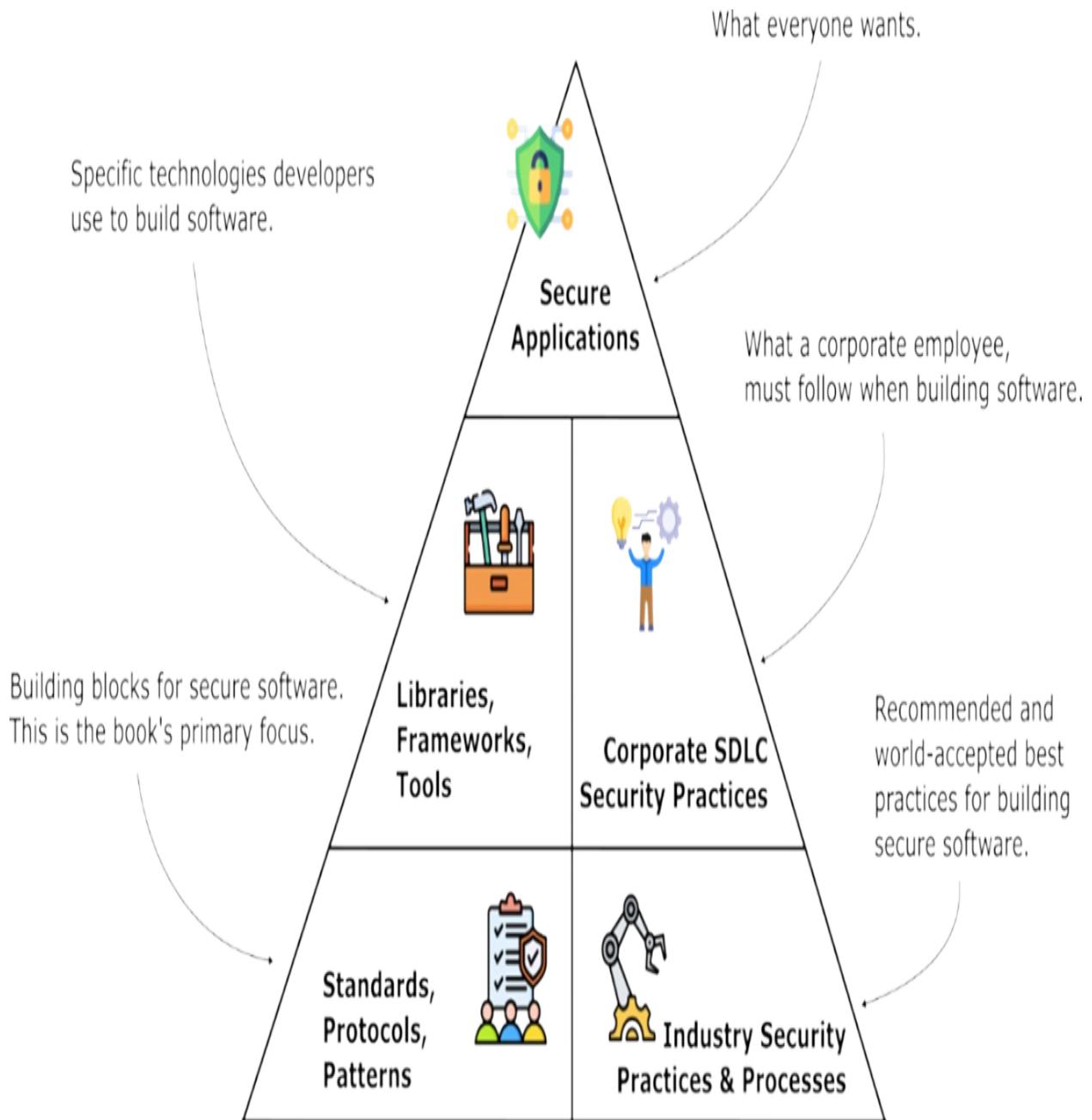
The heightened focus on security by senior business leaders affects

application developers in the following ways:

- Use all product security features: CISOs expect developers to use every security feature available in products to secure an application. Do you know how to configure and use the security features in the application server, database, object store, message broker, API gateway, service mesh, cloud services, programming language, and development frameworks being used on a project you are working on? It is no longer enough to know how to use a product, you must know how to use it securely.
- Follow corporate security standards: CISOs expect applications to pass strict corporate security assessments and audits. As a developer, you must explain to assessors and auditors how your application meets corporate security standards. This means you need to be able to speak the security language used by information security professionals so you can avoid costly remediation work to fix security issues late in the development cycle.
- Design and implement secure applications: CISOs expect architects and developers to design and implement secure applications. This means that you must be familiar with many security protocols and technologies required to design and implement secure applications.
- Enable DevSecOps Transformation: CISOs are investing heavily in breaking down the silos between the development, operations, and security teams. We call this set of practices DevSecOps. As a developer you need to become familiar with new tools, processes, and practices used to implement DevSecOps.

Figure 1.2 provides a map of the broad areas of application security. The top of the diagram above represents the goals of senior business leaders to build secure applications that can stand up to attacks. The higher layers of the diagram depend on the layers below them. To secure an application, you need to use security libraries; for example, a Java web application might use Spring Security to authorize user access.

Figure 1.2 Layers at the top depend on the layers below them. All the layers are required to produce secure application. The standards, protocols, and patterns used to secure applications are the primary focus of this book, they are the foundation that you need to use security libraries in your application effectively.

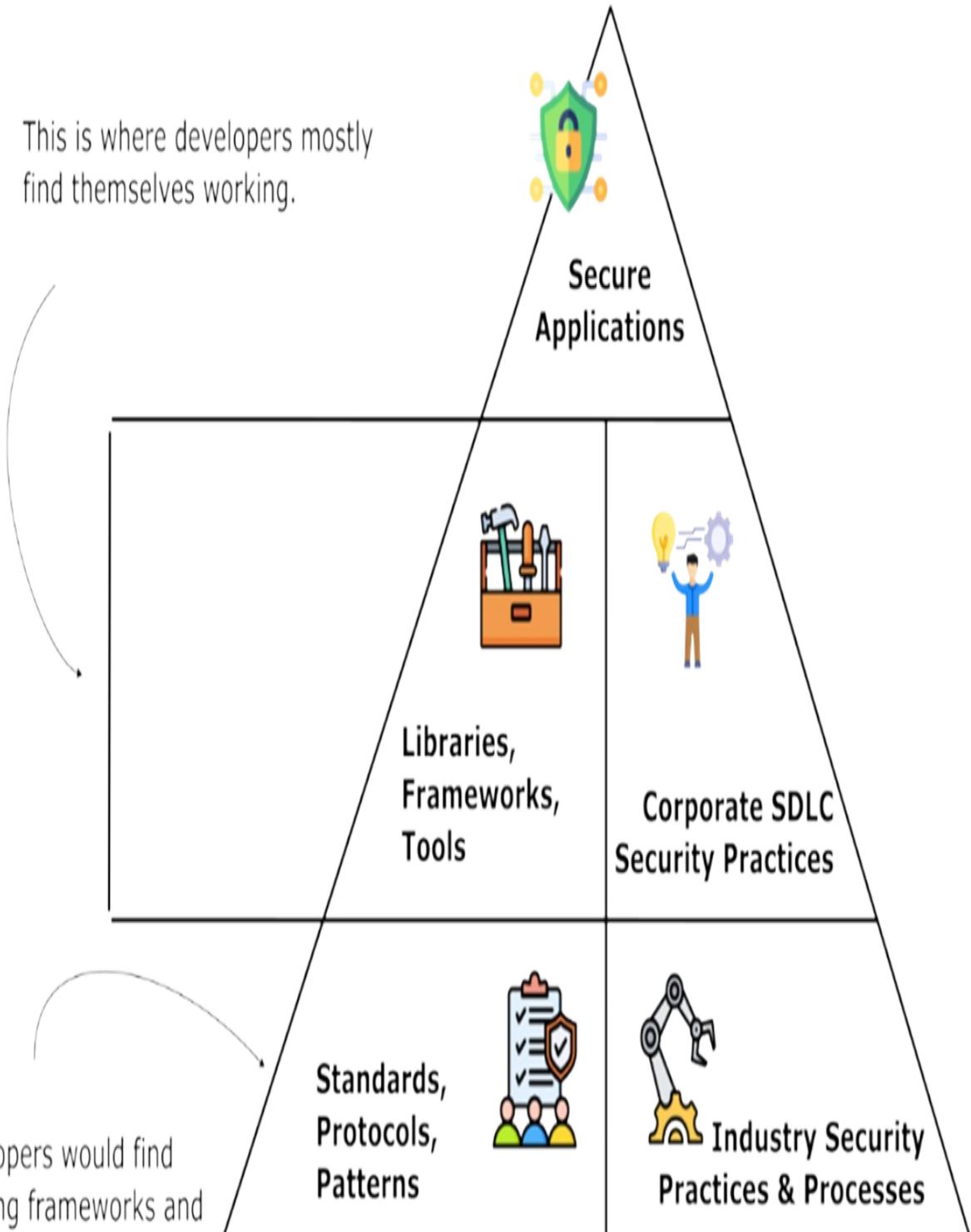


But security libraries are not enough to provide security. You must design, code, and maintain the application in a secure way by following the corporate security practices for application development, for example, performing a security code review or setting code analyzers that detect common security coding mistakes. As a developer, you spend your time in the middle layer of the pyramid above.

Security libraries and frameworks provide implementations of industry-

standard protocols and patterns in specific programming languages. For instance, the Java Standard Libraries include support for Transport Layer Security (TLS), which we'll explore in chapter 11, while Spring Security offers robust support for OpenID Connect (OIDC), discussed in chapters 12 and 13. Developers often invest significant effort into mastering these security libraries, which are critical to building secure applications. However, many developers find these libraries challenging to learn and cumbersome to use effectively (figure 1.3).

Figure 1.3 While developers often focus on libraries, frameworks, and tools at the mid-level, true security stems from foundational knowledge of standards, protocols, and patterns, as well as adherence to corporate and industry security practices. Bridging the gap between these layers leads to more effective and secure development.



Developers would find learning frameworks and libraries easier if they knew already the standards, protocols and patterns.

The root cause of developer difficulties using security libraries is a lack of

knowledge about the underlying standards, protocols, and patterns the libraries implement. If you understand the underlying security standards, protocols, and best practices, you will find security libraries and frameworks much easier to learn and use. For example, if you understand the OpenID Connect standard, you will find configuring Single Sign On authentication with Spring Security easy to configure and debug.

TIP

If you are familiar with the standards, protocols, and patterns, using security libraries and frameworks becomes much easier. Investing time to understand these underlying principles will significantly enhance your ability to work effectively with security tools.

This book focuses on teaching you the standards, protocols, and patterns implemented by the majority of application security libraries and frameworks through understanding the use case sample applications. The sample applications are implemented in Java using plain Java but also open-source frameworks such as Google Tink, Nimbus, Spring Security and others.

Applications are built on top of hundreds of open-source libraries and propriety software components. For example, today I am working on a Spring Boot application that depends on 106 open-source third-party libraries. I have seen some enterprise applications with 250+ library dependencies. Reusing software components across applications is a huge time and cost saver. However, it also introduces the possibility of catastrophic security failures.

In 2017, the American credit reporting agency Equifax was hacked, exposing the credit histories of 150 million American citizens. As of May 2019, Equifax has spent over 1.4 billion dollars on cleanup costs (<https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473>). The hack was caused by a known vulnerability (CVE-2017-5638) in the Apache Struts library that Equifax failed to patch even though the vulnerability was known for months (<https://www.zdnet.com/article/equifax-confirms-apache-struts-flaw-it-failed-to-patch-was-to-blame-for-data-breach/>). The Equifax hack illustrates the importance of keeping all components in an application's software supply chain updated.

In 2018, a backdoor for stealing Bitcoin was added to a popular JavaScript library called Event-Stream, which averages 2 million weekly downloads from the NPM repository (<https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/>). The backdoor successfully attacked the Copay Bitcoin wallet versions 5.0.2 to 5.1 (<https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/>). The attack occurred after the original author of the Event-Stream package got tired of working on it and transferred control of the project to a new developer who added the backdoor.

The Event-Stream attack is not an isolated incident. Many other attacks like it have been reported over the past few years. The Event-Stream attack illustrates the importance of vigilance against software supply chain attacks. Attacks against the software supply chain are increasing because they are highly effective and devastating.

To secure an application's dependencies, you must ensure that every direct and indirect dependency is secure. The whole dependency tree must be vetted & validated. Even though the application does not contain code that calls methods in certain dependencies, they must still be validated because it will be part of the application executable. Supply chain security is an industry-wide problem since every software producer depends on external code suppliers who in turn depend on other suppliers and so on. New security tools and processes must be built then adopted widely to secure the software supply chain.

You can easily detect vulnerable dependencies using an automated vulnerability scanner. The vulnerability scanner builds a list of all the dependencies an application uses by analyzing the application's code, building scripts, and generating artifacts. The scanner compares the application's dependency versions against a database of known vulnerabilities. If a match is found, the scanner will alert the development team.

The scanner should run on every code commit to alert the development team immediately if the commit introduces a vulnerability. Adding the scanner to the continuous integration pipeline so that builds with vulnerable

dependencies fail is a highly recommended best practice.

The scanner should be configured to rescan applications when the scanner's vulnerability database is updated with newly reported vulnerabilities. This is important because dependency vulnerabilities can be discovered after an application has been scanned and deployed to production.

The best vulnerability scanners can detect and automatically upgrade application dependencies. For example, suppose you have an enterprise Java application using Apache Struts version 2.3.25 containing the security vulnerability that was exploited in the Equifax hack.

With the scanners available now, the scenario described above is practical to implement. If you have fully automated test suites and continuous delivery pipelines, releasing a patch to production within hours of a CVE being disclosed is possible.

Recently, I ran into a mission-critical enterprise application using a 12-year-old version of a Java library because the developers wrote 100,000+ lines of business process code against internal APIs that were removed in a subsequent release. The organization put off upgrading the 12-year-old library for 10+ years due to cost and effort required.

Use only the publicly published interfaces so there is a documented path for upgrading to the next version whenever it gets released. Be kind to your future self and other developers who maintain the code that you are writing today by sticking to publicly published APIs for any libraries you are using; it is the right and secure thing to do.

GitHub offers a free vulnerability scanning service called Dependabot that you can turn on with the click of a button. However, many other vendors are offering excellent dependency vulnerability scanners. The vendor list constantly changes as start-ups release new products and existing companies acquire.

As an application developer, it is crucial that you appreciate the severity of the software supply chain security problem, keep up to date with advances in solutions, and push your employers to adopt these solutions as they become

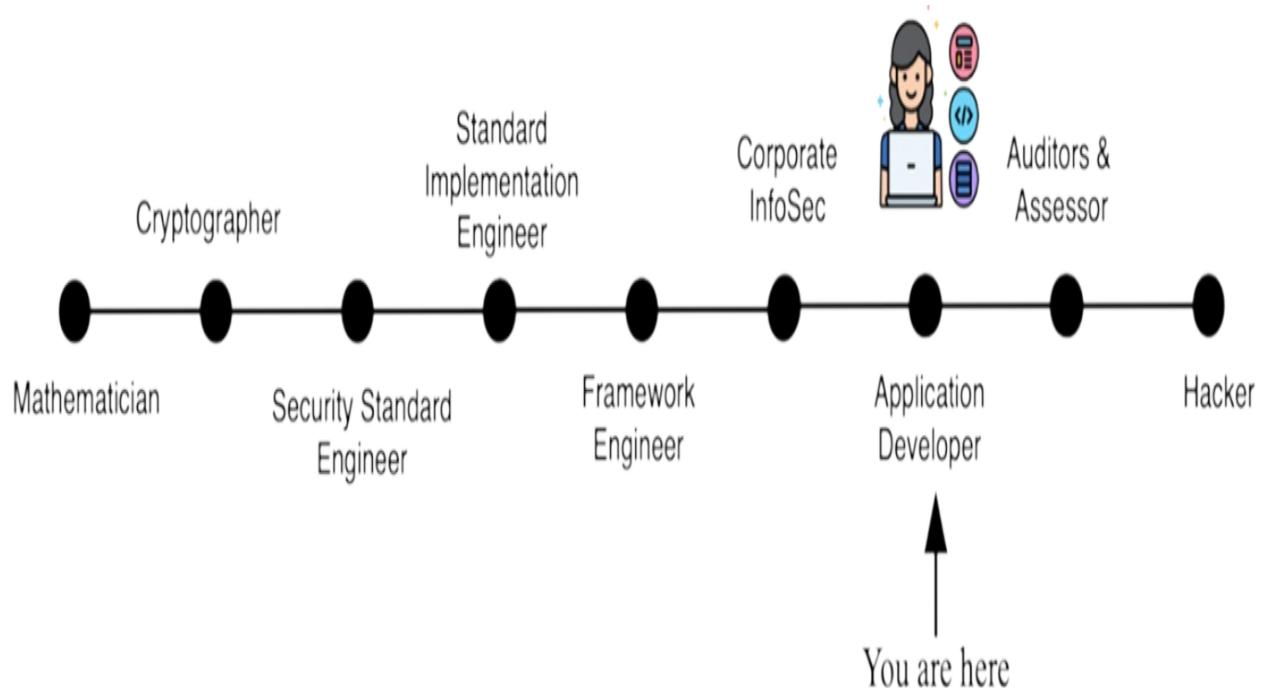
available.

The book cannot cover every software supply chain security tool on the market, but it will teach you the cryptographic primitives on which these tools are built. Part 2 of the book covers the foundational security technologies and standards used in all security areas, so make sure to read part 2 so you can more easily stay up to date with progress and developments in securing the software supply chain.

1.2 Roles and responsibilities in security

Computer security is a massive topic with many different subfields and specializations. It can take a lifetime to master computer security. As a developer, you must focus on the subset of computer security most relevant to your needs for writing secure applications. Figure 1.4 shows a continuum of security-related skills that can help you understand what you need to know and where to focus your efforts to learn security.

Figure 1.4 The spectrum of technical roles involved in computer security roles and responsibilities



Mathematicians produce the foundations upon which computer security rests. For example, elliptic curves are used to implement public key cryptography, frequently used in the TLS. Advances in mathematics can make new security algorithms possible or break existing ones. As a developer, you do not need to be familiar with the mathematics underlying computer security.

Cryptographers use specific areas of mathematics to design foundational algorithms for encryption, secure key (password) exchange, hashing, random number generation, digital signatures, and secure combinations between systems. Cryptographers also analyze security algorithms and protocols for weaknesses. As a developer, you don't need to understand how cryptographic algorithms work, but you need to understand what the algorithms do, when to use them, and how to configure them. Part 2 of the book covers foundational algorithms that you should be familiar with as a developer.

Security standard engineers define security standards that allow applications to interoperate across network boundaries, operating systems, and programming languages. Transport Layer Protocol (TLS) and OpenID Connect are standards you will frequently encounter as a developer. It is essential to be familiar with the security standards so you can:

- Write secure applications that pass corporate security audits
- Quickly configure libraries and frameworks that implement the standards without spending hours searching blogs and online resources such as stackoverflow.com
- Debug security issues and configurations quickly

Standard implementation engineers implement security standards as reusable libraries in a variety of programming languages. For example, OpenJDK engineers implement a large number of security standards as part of the Java standard libraries. As an application developer, you need to learn how to use libraries that implement security standards correctly. The book provides Java-based implementations for the standards we will study.

Framework engineers implement libraries to accelerate application development in a specific programming language. Framework developers focus on common use cases that applications need to implement and provide out-of-the-box code to implement the functionality. As an application

developer, you need to master these frameworks to build applications effectively. The book uses the Spring Framework ecosystem to demonstrate how to implement a variety of common security use cases.

Corporate Information Security is the team responsible for the security of all applications deployed in a company. They write the corporate security standards that applications must adhere to and consult with development teams to help them secure their applications and comply with corporate standards. As a developer, you need to become familiar with your employer's security standards; this book provides you with the technical background to quickly understand corporate security standards. A key goal of the book is to help you become a better partner to your InfoSec team.

Auditors and assessors evaluate the design and implementation of applications to ensure that they comply with security best practices and corporate standards. Information Security audit/assessment is a prerequisite for getting an application to production. Failing the security audit means a delay in releasing to production. A key goal of this book is to teach you security skills so that your applications pass security audits and assessments without costly length rework.

Hackers break into systems to steal data, money, and products. This book does not cover the numerous techniques that hackers use to break into applications.

As you can see, computer security brings together many roles, each with its own responsibilities and expertise. You don't need to become a mathematician, a cryptographer, or an auditor to write secure code. But you do need to understand enough of their work to use it wisely. This book focuses on the developer's part of the spectrum: learning how to apply standards, use libraries, and build applications that meet both technical and corporate security expectations. With this foundation, you'll be better equipped to collaborate with security teams, pass audits with confidence, and most importantly, create software that users can trust.

1.3 What you will learn in this book

This book is for developers who are just starting out or already have some experience. We'll begin with a clear, big-picture view of what security means in real software projects, so you understand why it matters. Step by step, we'll then dive into the basics of cryptography and explore the protocols and standards that bring those ideas to life. By the end, you'll not only know how these pieces fit together but also feel confident applying them to build safer and more reliable applications.

The book's examples are provided with Java and Java-related libraries, but the skills you learn from the book are invaluable regardless of the language and platform you use. You're invited to read the book even if you don't declare yourself a Java developer as long as you understand standard OOP language instructions.

When you finish the book, you will have learned the following skills:

- Understanding what security means in real-world software development.
- Recognizing common risks and how attackers might exploit them.
- Applying the basics of cryptography to protect data.
- Using protocols and standards (like TLS and OAuth) in practice.
- Designing applications with identity, authentication, and authorization in mind.
- Building secure communication between services.
- Knowing how to spot and fix common security mistakes in code.
- Gaining confidence to make design choices that keep applications safe.

1.4 Summary

- Security vulnerabilities can exist at every layer of the stack, from hardware (e.g., Meltdown, Specter) to application code.
- Security is everyone's responsibility, not just InfoSec teams - developers play a central role.
- The business impact of breaches is massive (e.g., Marriott, Equifax), often costing millions or even billions.
- CISOs expect developers to:
 - Use all product security features
 - Follow corporate security standards

- Design and implement secure applications
- Embrace DevSecOps practices
- Security libraries (like Spring Security) are essential but hard to use unless you understand the underlying standards and protocols.
- Supply chain attacks (e.g., Equifax Apache Struts, Event-Stream Bitcoin theft) highlight the need for vigilance in managing dependencies.
- Automated vulnerability scanning in CI/CD pipelines is a best practice to detect and fix issues quickly.
- Stick to published APIs in libraries to ensure maintainability and security over time.
- Different roles contribute to security: mathematicians, cryptographers, standards engineers, framework engineers, InfoSec teams, auditors, and developers.
- Developers don't need deep expertise in all these roles, but they must understand enough to apply standards and use libraries correctly.
- This book teaches developers the foundations (cryptography, protocols, standards) so they can confidently build secure, reliable applications.

2 Standards for implementing authentication

This chapter covers

- Analyzing customer, employee, and partner preferences for authentication
- Discussing standards to enable secure user authentication
- Identifying the technologies for securing sensitive application credentials

All applications, whether a million-line monolith or a thousand-line microservice must solve the following four security problems:

- Securing communication channels
- User authentication
- Handling sensitive credentials such as API keys required to access external services
- Running the application securely in a cloud environment

We have explored how to secure communication channels with TLS, basically locking the door to your data so no one sneaks in. But what good is a locked door if you're handing out keys to just anyone? That's where authentication comes in.

Now, we'll dive into the tools and methods for solving authentication challenges. Why? Because you need to make sure the person knocking at your app's front door isn't a hacker in disguise, or worse, your ex trying to get into your Netflix account.

Think of authentication as the bouncer at a club. You wouldn't want a bouncer who lets everyone in without checking IDs, would you? (Unless your app is a sketchy underground party, in which case... you do.) By understanding the authentication landscape, you'll learn how to be that

bouncer who knows who belongs, who doesn't, and who's using a fake ID.

And trust me, mastering these tools is essential. Because in the world of cloud-native apps, if you don't know how to authenticate users properly, you're not just leaving the door open. You're throwing a 'Hackers Welcome' party with free snacks. Let's make sure your app is secure and smarter than your average bouncer. Ready to level up

2.1 Logging users in

Applications accessed over the network require users to login so that the application can ensure a user is only able to access the data and functionality they are authorized to access. Most commonly logging into an application means entering a username and password combination (figure 2.1).

Figure 2.1 An authentication form, commonly referred to as a login form, serves as the primary interface through which users enter their credentials to authenticate and gain access to the system.

The simplest authentication method involves providing a set of credentials consisting of a username, which uniquely identifies the user within the system, and a password, which is a secret known only to the authenticated user

log in

email address

password

log in now

[forgot your password?](#)

Passwords are easy to implement in an application, but they are terrible for security and user experience for the following reasons.

- *Weak passwords* Humans have a hard time remembering long, complex passwords, so many people use weak passwords that are easy to remember.
- *Password reuse* I have 308 online services that have asked me for a username and password. I cannot remember 300+ unique username/password combinations; therefore, I use a password manager to generate complex unique passwords for each site. A majority of users do not use a password manager, so they reuse the same username/password across many online services. When a password leak

occurs on some online service, hackers use the leaked passwords to compromise user accounts on unrelated services.

- *Storing passwords is hard and expensive* Storing passwords securely on the server side is a complex problem requiring deep security expertise to implement correctly. Even well-funded Silicon Valley companies get password storage wrong. For example, LinkedIn^[1] leaked 6.5 million passwords in 2012; worse, the leak was not noticed until 2016. It is not enough to implement password storage securely using today's best practices. You must keep up to date with advances in attacks. If you are storing passwords on the server side, you must be ready to invest time and money to keep those passwords secure.

You can easily avoid storing passwords in your application by using a Single Sign On (SSO) service via an industry-standard protocol such as OpenID Connect. SSO services solve many challenging problems beyond just password storage as we'll discuss in chapter 12.

Consider ACME Inc. the shoe retailer with 1000 physical stores in multiple countries and an online shopping application that we discussed previously. Customers can buy shoes from ACME in the stores, or online using a web application or native iPhone and Android mobile apps. ACME has three categories of users that need access to its systems: customers, employees, and partners. Each user type has different authentication preferences and needs, which will examine to understand what capabilities an SSO service must provide us with.

2.1.1 Customer authentication

Like all online stores ACME Inc. wants to minimize friction in the checkout process to maximize sales. Returning customers should be able to quickly log in to place an order. They should also be able to use a comfortable authentication method of their choice (figure 2.2). New customers should be able to quickly provision a new account for a fast checkout process when they return to shop for more shoes.

Figure 2.2 Customer authentication preferences. Some customers want to use biometric authentication features, such as fingerprint scanners and face recognition, on their devices and phones. Some customers want to be able to login using their existing accounts with large online

service providers such as Google, and Facebook. Some users want to be able to login using a traditional username and password combination. Single Sign On services can accommodate all these types of authentication preferences and more.



I want to log in using the fingerprint scanner on my laptop.



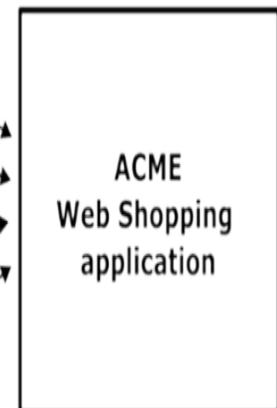
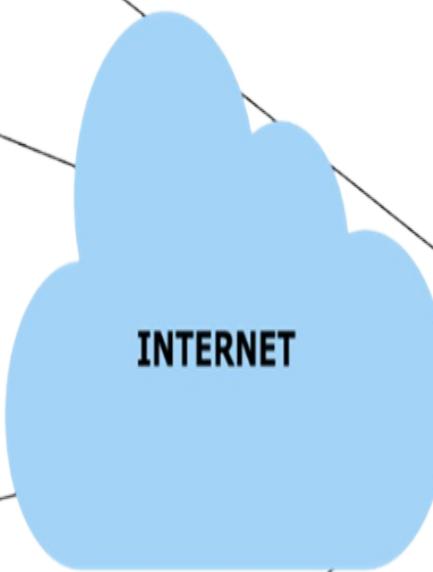
I want to log in using my Facebook or social network account.



I want to log in using the face recognition feature of my mobile phone.



I want to log in using a username and a password.



We'll discuss three main approaches that can be used to implement on-demand account provisioning and login for customers.

- *Use a third-party account via OAuth2 or OpenID Connect* Many customers have accounts with popular online services such as Google, Microsoft, Facebook, Twitter, GitHub ... etc., which they can re-use with ACME Inc. Reusing existing accounts saves the user from having to create a new username/password combination, and it shifts the responsibility for protecting the user's account from hackers to large organizations with more resources and expertise. OAuth2 and OpenID Connect are industry standard, widely supported protocols that enable users to use existing accounts across online services. Chapters 12 and 13 of the book cover OAuth2 and OpenID connect, showing you how to use them for logging users into apps.
- *Passwordless biometric login with WebAuthn* Modern smartphones and laptops allow users to unlock them using facial recognition or a fingerprint scan. Logging into a personal device with biometric scan is very convenient and popular. ACME Inc. customers want to create an account and log into it using the biometric capabilities of their device to access the online shopping site. WebAuthn is a new industry standard for web authentication using device biometrics standards. It is widely supported in the Apple and Android ecosystem and desktop web browsers. You can try out the WebAuthn user experience at <https://webauthn.io>. Leverage the WebAuthn protocol to eliminate passwords from the login process. We will cover WebAuthn in chapter 14.
- *Multifactor Username / Password authentication* If users don't have a device that supports WebAuthn or an account on a social network that they want to reuse, you can provide a fallback to username / password-based authentication. As a best practice, you should give users the choice to use multifactor authentication using authenticator apps or SMS codes with password-based authentication. We will cover multifactor authentication technologies in part 5 of the book.

It is possible to implement social login using OpenID Connect, biometric login using WebAuthn, and multifactor password-based login directly in a

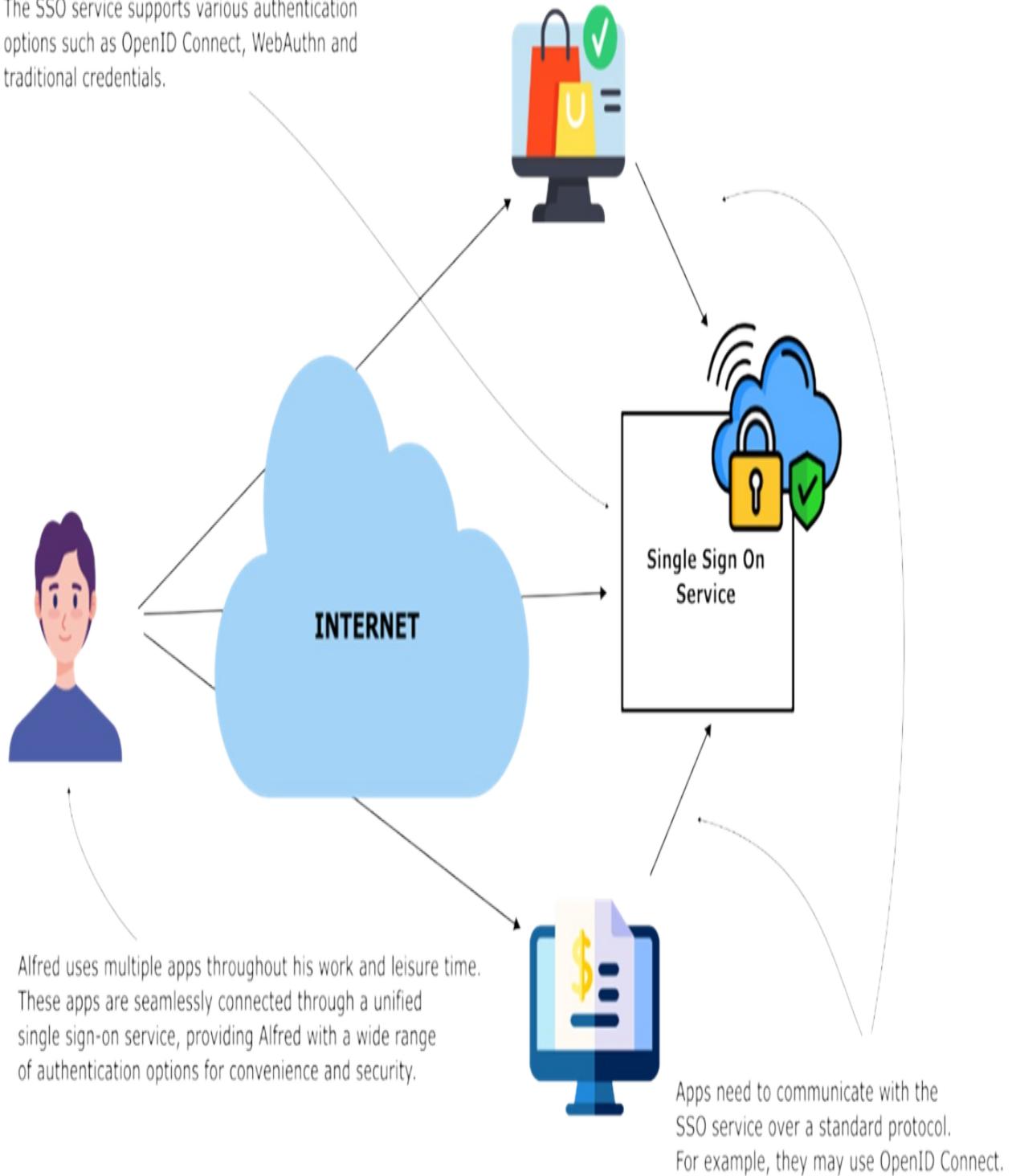
monolithic application using popular libraries and frameworks. For example, a Spring Boot Java application can use Spring Security to implement authentication directly in the application.

However, most companies have multiple monolithic applications. Implementing security in each application is expensive, time-consuming, and leads to interoperability issues. It is a best practice to externalize authentication into a Single Sign On (SSO) service as we'll discuss in chapters 12 and 13.

Figure 2.3 illustrates how Alfred (a normal user) utilizes multiple applications for both work and leisure. These applications seamlessly integrate with a single sign-on (SSO) service, enabling Alfred to authenticate effortlessly through various login options.

Figure 2.3 Single Sign On (SSO) Service handles user account creation and authentication. Multiple applications can use the same SSO service, simplifying security for application developers. The SSO service implements support authentication using OAuth2, OpenID Connect, WebAuthn, and multi-factor password-based authentication.

The SSO service supports various authentication options such as OpenID Connect, WebAuthn and traditional credentials.



Applications delegate user account creation and authentication to the Single Sign On (SSO) service. Applications can interact with the SSO service using industry-standard protocols; OpenID Connect is the most popular modern protocol, but other protocols, such as SAML (Security Assertion Markup

Language), can be used. There are three ways to get access to an SSO service.

- *Fully managed SaaS SSO service* Many companies offer fully managed cloud-based SSO services. For example, Okta, VMware, Amazon, Google, Microsoft are leading providers of SaaS based SSO services.
- *Self-run off-the-shelf SSO service* If you want to run your own SSO service, you can use commercial products such as Workspace One, Ping Federate, CA SiteMinder, ForgeRock, or an open-source project such as Keycloak, Dex Idp.
- *Self-built and run SSO service* If you have business and technical needs that cannot be addressed by using a SaaS solution or an off the shelf packaged solution you can build and run your own SSO service on top of popular open-source libraries.

The choice between fully managed SaaS, off the self-solution and a custom built SSO service is based on variety of technical and business requirements. Ideally you want to avoid having to build your own service as that can be quite complex, using a SaaS, commercial or open-source solution is the best path forward for the majority of organizations.

As a developer getting comfortable with OAuth2, OpenID Connect, WebAuth2, and the technologies for multi-factor password-based authentication, provide a complete toolbox for all your authentication needs today.

Part 5 of the book shows you how to build a custom example SSO service that implements OAuth2, OpenID Connect, WebAuthn, and multi-factor password-based authentication using the Spring Security Authorization Server as a way to deepen your understanding of SSO technologies.

2.1.2 Employee authentication

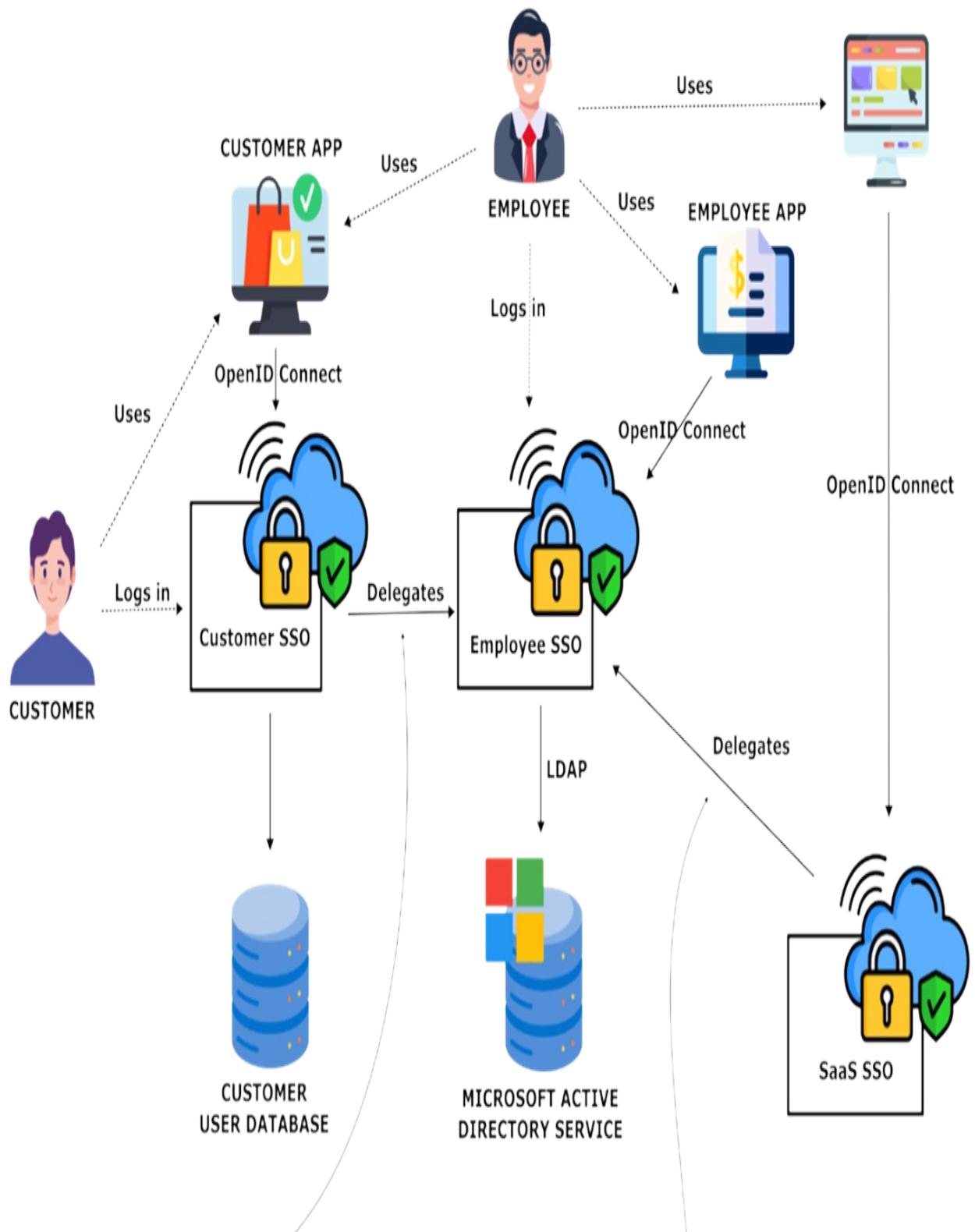
Let's get back to the ACME Inc. scenario, where you're in charge of setting up and managing security for the organization. ACME Inc. operates 1000 retail stores across five countries, each store has two Windows desktop computers that employees use to manage the store location.

ACME's corporate headquarters has 250 employees who use corporate phones, tablets, and laptops (Windows and MacOS). Like many enterprises, ACME uses Microsoft Active Directory (AD) to authenticate employee's login into desktops, laptops, and all corporate systems. Employees need to access the following types of applications.

- *Internally deployed commercial off the shelf applications* These apps are built by vendors who have added Active Directory (AD) into their products, allowing ACME's system administrator to configure AD as the authentication mechanism for these internal apps.
- *External cloud-based SaaS applications* ACME employee uses many SaaS services such as JIRA for project management software, Slack for chat ... etc. Employees want to be able to log into these remote SaaS applications using their own corporate-issued AD credentials. Most SaaS applications bill ACME on a per-user basis, so ACME wants to be able to control which employees have access to these external SaaS services based on how they are set up in Active Directory. When an employee leaves ACME Inc., their external SaaS-based account should be de-provisioned automatically.
- *Internal-only employee-facing custom-built applications* ACME Inc. has several internal custom-built employees-facing applications. Employees want to gain access to these applications using their existing AD credentials.
- *Employee-only interfaces on customer-facing applications* The online shopping application used by customers has several screens that only employees can access. How can the online shopping application enable customers to log in with customer accounts and employees to log in with their Active Directory accounts?

The Single Sign On (SSO) service we introduced in the previous section (figure 2.3) can be used to allow employees to access internally deployed commercial off-the-shelf applications, external SaaS based applications, internal custom-built applications, and the employee-facing interfaces on customer-facing applications.

Figure 2.4 A corporate Sige Sign On (SSO) service makes it possible for employees to access customer facing apps, internal employee only apps, and external SaaS apps using a single set of credentials



The customer SSO detects the user is an employee and forwards the login to the internal SSO, granting access to employee-only features.

When accessing a SaaS app, the provider's SSO redirects login to the employee SSO, letting the user log in with corporate credentials.

Companies with more than a handful of employees must implement access controls to corporate systems. When employees are hired, they must be granted access. When employees quit, access must be revoked. An employee SSO service is critical for the functioning of a modern company. If you are building an employee-facing application, you have to integrate it into the corporate SSO service. What protocol should this integration use?

Modern corporate SSO services support OpenID Connect (OIDC), which you can use to secure employee-facing apps. If the corporate SSO service does not support OIDC, you can deploy a bridge from OIDC to the protocol supported by the SSO service. For example, an OIDC to SAML bridge allows your application to use OIDC and the SSO service to use SAML.

The LDAP and SAML protocols have been around for decades. They are widely deployed in the enterprise. This book does not cover SAML and LDAP as they are well covered in other books. We focus on OpenID connect because it can be used for both customer-facing and employee-facing apps, and it is much easier to work with than LDAP and SAML.

2.1.3 Partner authentication

ACME Inc. has several partners that need access to its internal systems to assist ACME Inc.'s employees. For examples, the vendor technical support might need access to internal ACME systems to help them during an upgrade, the company is reliant on external vendor that provides customer support. There are two ways that ACME Inc. can grant partners access to its internal systems:

- *Provide partners with employee account.* Partner employees are provided user accounts on ACME Inc.'s employee directory. This approach is commonly used because it is straightforward to implement. If a partner employee is fired or quits, ACME's IT staff must be notified to revoke access to the partner employee. The notification mechanism can be slow and that introduces a risk where a partner ex-employee still has access to ACME Inc.'s systems when they should not.
- *Delegate partner employee authentication to the partner's employee*

SSO service. In this approach ACME Inc. configures the ACME employee SSO service to recognize when a partner employee is trying to login, and then delegate the actual login to the partner's SSO service. This way as soon as a partner employee loses access to the partner's SSO service they will also lose access to ACME's Inc's systems.

2.1.4 Phishing resistant authentication

Hackers can break into systems by attacking machines or the humans using the machines. When attacking machines, hackers look for technical vulnerabilities in applications and infrastructure that they can exploit to break in. Alternatively, hackers can trick human users with legitimate access to targeted systems into giving them access to the target system or performing an action they are not supposed to. Attacks targeting human users are called phishing attacks, and they are growing rapidly because they are highly effective.

For example, consider the following steps that hackers can use to break into ACME Inc.'s cloud infrastructure. Using a LinkedIn search, hackers identified Joe Smith as the lead cloud DevOps engineer on ACME Inc.'s online retail team and said that he can be reached at jsmith@acme.com. Through some online sleuthing, the hackers determined that ACME uses Google Cloud to run its online services. Since Joe is ACME's lead DevOps engineer, hackers guess that Joe must have administrator access to the GCP console.

The hackers craft an email that looks like it was sent from Google Cloud, informing Joe that there is some issue in the ACME Inc. systems and that he should click a link in the email to get more details and resolve the issue.

Joe has been working for 8 hours with no breaks, he is very tired, so he does not notice that this is a very convincing but fake email. Joe clicks on the link in the email, which takes him to a login screen replica of the Google login screen. Joe does not notice that the URL in the browser address bar is not a Google URL. He types his username, password, and the one-time password from the authenticator app into the fake login form.

The fake login form captures Joe's credentials and uses them to log into Google Cloud, then sends Joe to a screen informing him that the email he received was sent in error. Joe is happy; he has no extra work to do; he quickly closes his browser and heads home, unaware that he has just given hackers access to ACME Inc.'s GCP infrastructure. Over the next few hours, hackers use Joe's administrator credentials to steal critical data from ACME's systems.

Phishing attacks are super effective because they target humans using a computer system. Even the most security-savvy and careful system administrator can be phished by a determined attacker. You can make your application resistant to phishing attacks like the one outlined above by allowing users to login physical security keys such as a Yubikey.

Figure 2.5 A user has to insert the Yubikey in their laptop, then press the button when on the login screen for a an application that support a physical security key. The Yubikey will check that the URL of the site the user is trying to log into matches the URL stored inside the Yubikey. If the URLs don't match the login will fail. A Yubikey can protect against phishing attacks such as the one described in the text above.



Phishing-resistant physical security keys are gaining in popularity and should be used to protect high-value user accounts. Adding support for phishing-resistant authentication using physical security keys is easy for keys that support the WebAuthn standard, including Yubikey. WebAuthn will be covered in Part 5 of the book.

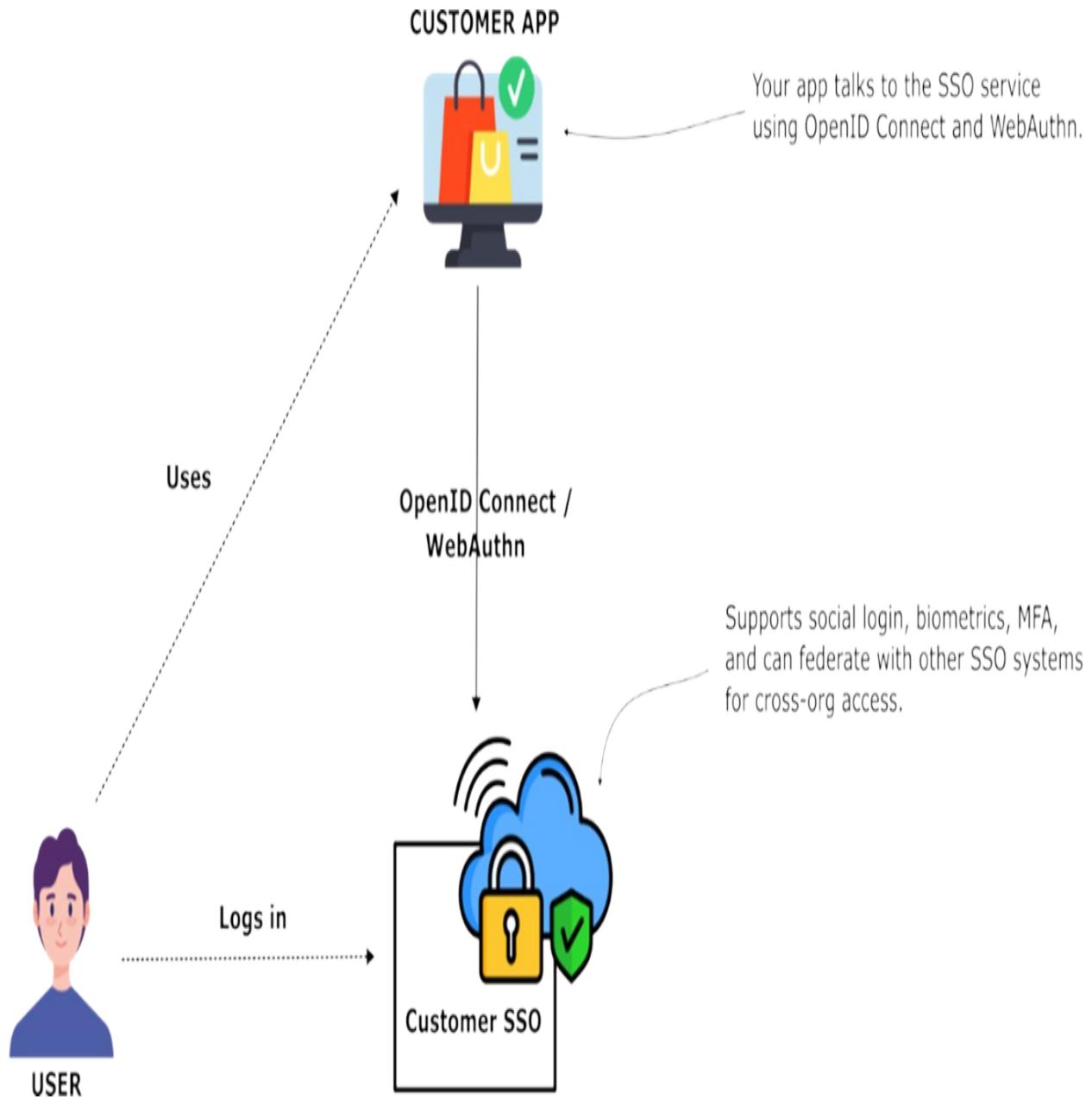
2.1.5 Authentication technology from a developer's perspective

User authentication is one of the first security features developers add to applications. Externalizing user authentication from a monolithic application into a Single Sign On (SSO) service is the recommended approach (figure 2.6). Applications interact with SSO services using industry-standard protocols. For a developer, the most important protocols to be familiar with are OAuth2, OpenID Connect, and WebAuthn.

OAuth2 and OpenID Connect enable applications to delegate login to an external service, thus eliminating the need for an application to manage and store usernames and passwords. All the major social networks provide support for OAuth2 and OpenID Connect, allowing users to reduce the number of passwords they need to remember. We'll discuss this subject more in detail in chapters 12 and 13.

The web authentication protocol WebAuthn can be used as an alternative to passwords. Applications can leverage WebAuthn to allow login with biometric scanners such as facial recognition or fingerprint scanners, as well as phishing-resistant physical security keys such as a Yubikey.

Figure 2.6 Externalize application authentication into an SSO service that applications can access using OpenID Connect and WebAuthn. If you know OpenID Connect and WebAuthn, you can use all modern SSO services.



OpenID Connect and WebAuthn are supported via high-quality libraries and frameworks in all programming languages. Part 5 of the book provides you with a Spring Security-based set of sample applications that you can use to learn the protocols.

Mastering OpenID Connect and WebAuthn enables you to write applications that work with all modern SSO servers. However, you need to be familiar with all the content of Part 2 of this book, especially TLS, JOSE, X.509 certificates, and public key infrastructure. So please take the time to go

through the foundational topics covered in part 2.

BEST PRACTICE

Use a Single Sign On (SSO) service. It will make your application more secure, and it is easier than building your own authentication system. Acquire a SaaS, commercial, or well-maintained open-source SSO service rather than building your own. Keep your SSO service patched and up to date.

Roles and Responsibilities

Understanding the roles and responsibilities of different team members is crucial for successfully implementing Single Sign-On (SSO) services in your organization. This guide breaks down the key duties of information security engineers and developers, making the process more transparent and manageable.

- Information Security Engineers
 - Decision Making - Information security engineers decide which Single Sign-On (SSO) service can be used within the company.
 - Setup and Configuration - They are responsible for setting up the SSO service and determining which configurations of OpenID Connect and WebAuthn to enable.
- Developers
 - Implementation - Developers must integrate OpenID Connect and WebAuthn libraries into their applications to enable interaction with the chosen SSO service.
 - Protocol Knowledge - Understanding OpenID Connect and WebAuthn protocols is essential. Once the protocols are mastered, using a programming language-specific library becomes straightforward. Focus on learning the protocols first to simplify implementation.

Learning the protocols might seem daunting at first, but once you understand their principles, implementing them in your applications will feel like a breeze.

2.1.6 Exercises

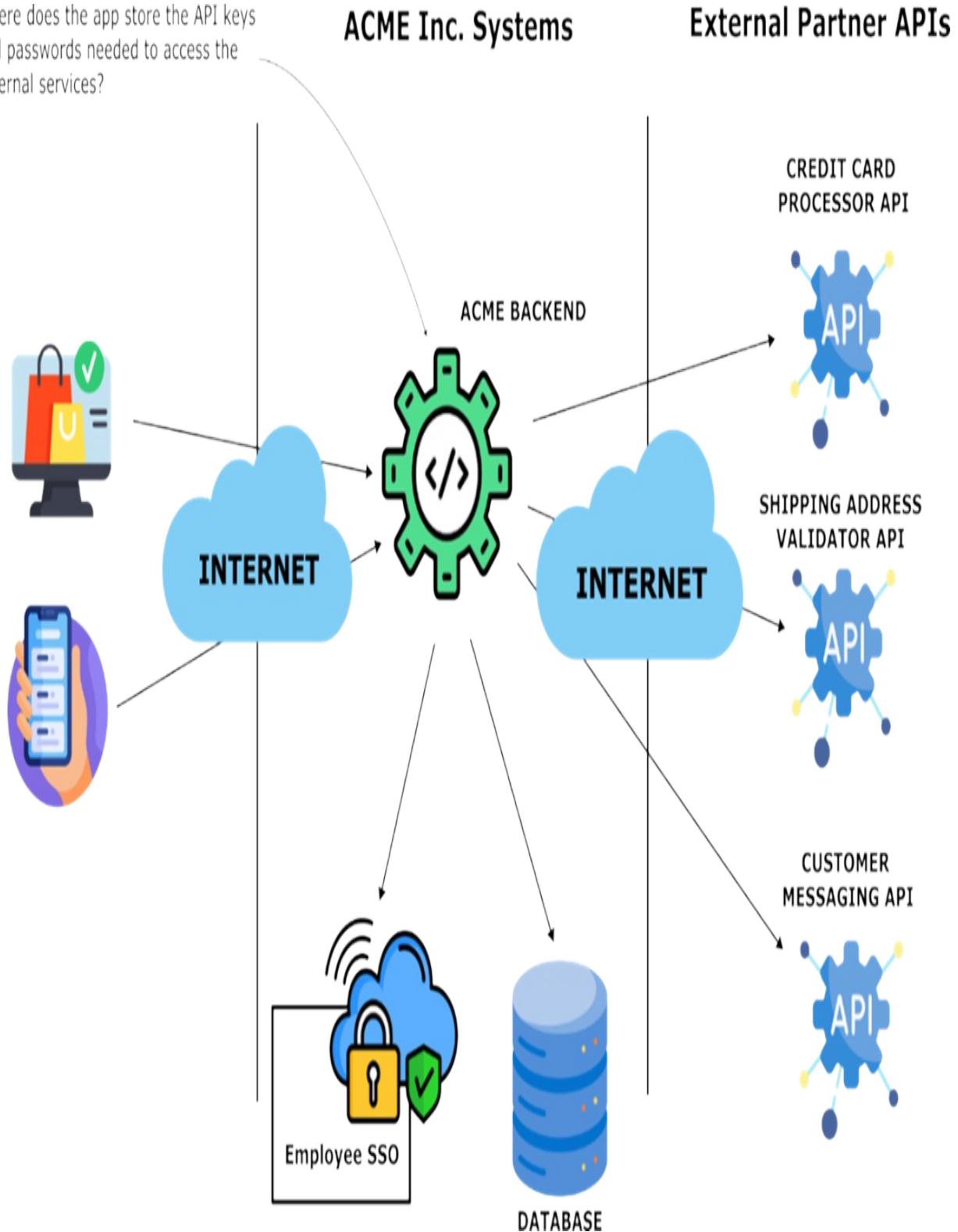
1. Why are passwords terrible for security and user experience?
2. What protocol makes it possible to log in without a password?
3. What is OpenID Connect used for?
4. What is a phishing attack?
5. Which authentication technology can protect against phishing attacks?
6. What capabilities does an SSO service provide?
7. Should every application use an SSO service?

2.2 Securing application credentials

Monolithic applications and microservices must access database servers, message brokers, email servers, and internal and external APIs to implement the application functionality. For example, ACME's online shopping application depends on the services in figure 2.7.

Figure 2.7 Applications depend on internal and external services that require passwords and API keys. The credentials to access services are extremely sensitive and must be protected. How can an application store and access sensitive credentials it needs to operate?

Where does the app store the API keys and passwords needed to access the external services?



When a customer buys a pair of shoes on ACME's website, the backend makes an API call to a payment processor to charge the customer's credit

card. The credit card processing API requires an API key to authenticate that the call comes from ACME's systems. If hackers get access to the payment API key, they can cause financial damage to ACME Inc.

Where can the application store the payments API key so that it is only accessible to the application? The payment processor API keys are only valid for one month from the date they are issued. How can ACME implement a reliable process for regularly updating API keys in production servers?

Generalizing beyond API keys, applications require a generic way to store the secrets they need to operate. Storing secrets in configuration files is a very common practice; however, it suffers the following serious drawbacks.

- *Configuration drift*, a highly available application running on ten servers, requires a copy of the configuration file on each server. An update process can fail easily, resulting in two out of ten servers having the wrong configuration settings.
- *Hard to secure*, operating systems do not provide the fine-grained security controls required to properly lock-down configuration files containing sensitive secrets.
- *Difficult to audit*, filesystems do not provide enough fine-grained audit trail to investigate security incidents involving stolen credentials.
- *Hard to regularly rotate credentials*, changing credentials on a regular basis is a security best practice. This means that applications must be able to determine the expiry dates for credentials or be notified when credentials are updated. Implementing credential rotation with configuration files is very difficult and error-prone.

Putting all application secrets in a centralized credentials service is a great alternative to using configuration files (figure 2.8). The primary benefits of using a credential service are:

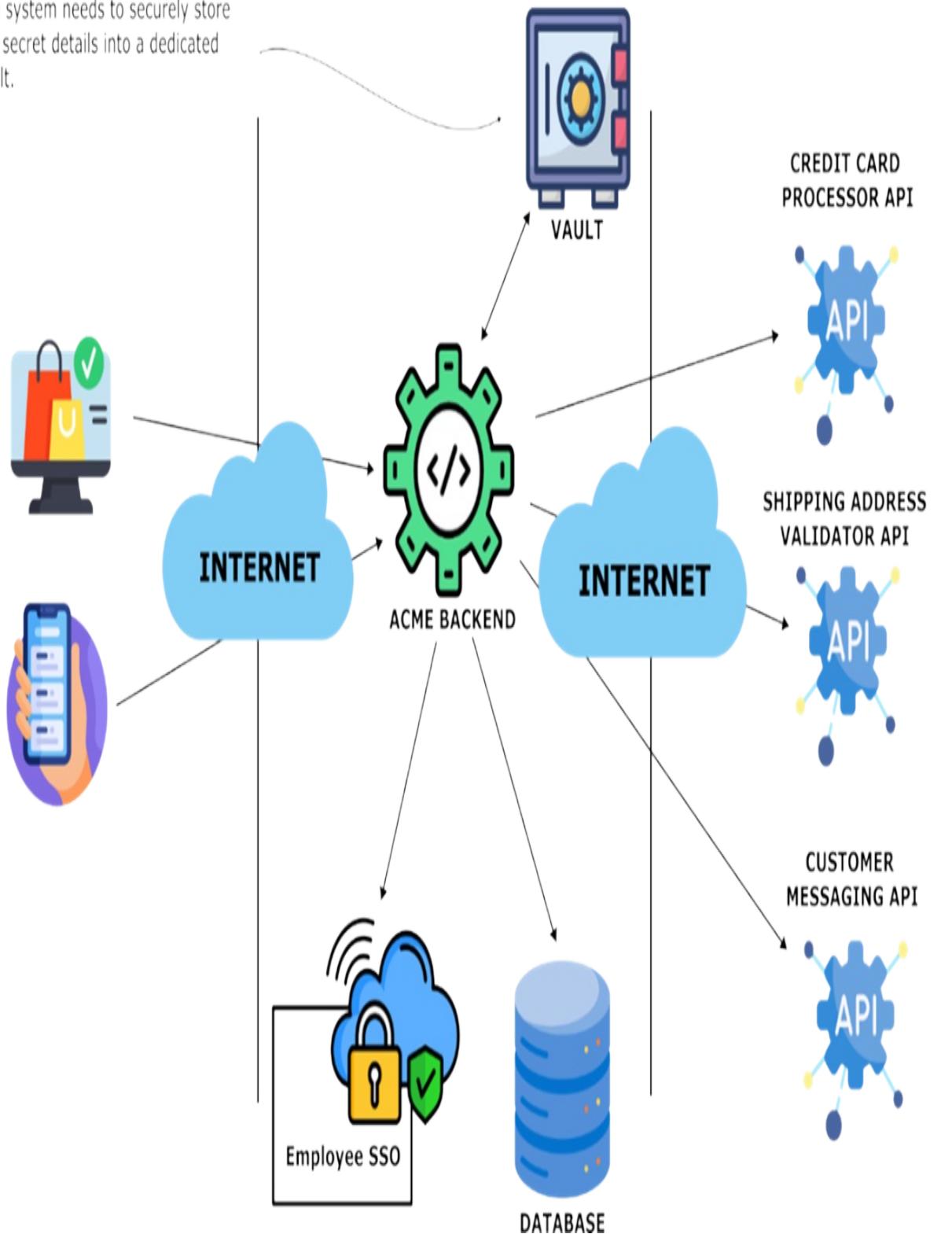
- *One source of truth* Centralized credential services act as the source of truth for all application instances, eliminating configuration drift caused by files.
- *Easy updates* Once credentials are updated in the credential service, applications can be notified about the updated value, enabling them to use new values without needing to restart.

- *Simplified credential rotation* Credential services store metadata about credentials, such as expiry times, allowing applications to choose the currently valid versions of credentials.
- *Comprehensive audit logs* Credential services offer fine-grained audit logs that are essential for proactive monitoring of suspicious activity and investigating security incidents.
- *Hardware security module support* Credential services typically use specialized hardware security modules to ensure that their secrets are well protected against even the most determined attackers.

Figure 2.8 A credential vault is a centralized store of all sensitive configuration values, such as passwords, API keys, digital certificates, and other secrets the application needs to access at runtime.

ACME Inc. Systems

The system needs to securely store the secret details into a dedicated vault.



Do not write your own credential service. Implementing and maintaining a credential service requires deep security expertise, and mistakes can cause catastrophic security failures. There are three common approaches to getting access to a credential store.

- *Cloud provider credentials service* Public cloud providers offer centralized credentials management services for applications running on the public cloud, too. For example, an application running on Azure can use the Azure Key Vault service to manage all sensitive credentials. The book covers AWS Key Management Service, Azure Key Vault, and GCP Key Management Service in part 4.
- *Container platform credentials service* If the application runs on a container platform, it can use its built-in credential service. Applications running on Cloud Foundry can use CredHub, a credential store built into the Cloud Foundry platform. Kubernetes provides a fundamental credential storage mechanism called Secrets Map, which we will cover in part 3 of the book.
- *Self-deployed credential service* There are many commercial and open-source credential management services that you can deploy and run as a service in your infrastructure. For example, Vault is a widely deployed open-source credentials management service with a commercial enterprise version offered by HashiCorp the company that created Vault. We will cover HashiCorp Vault in part 4 of the book.

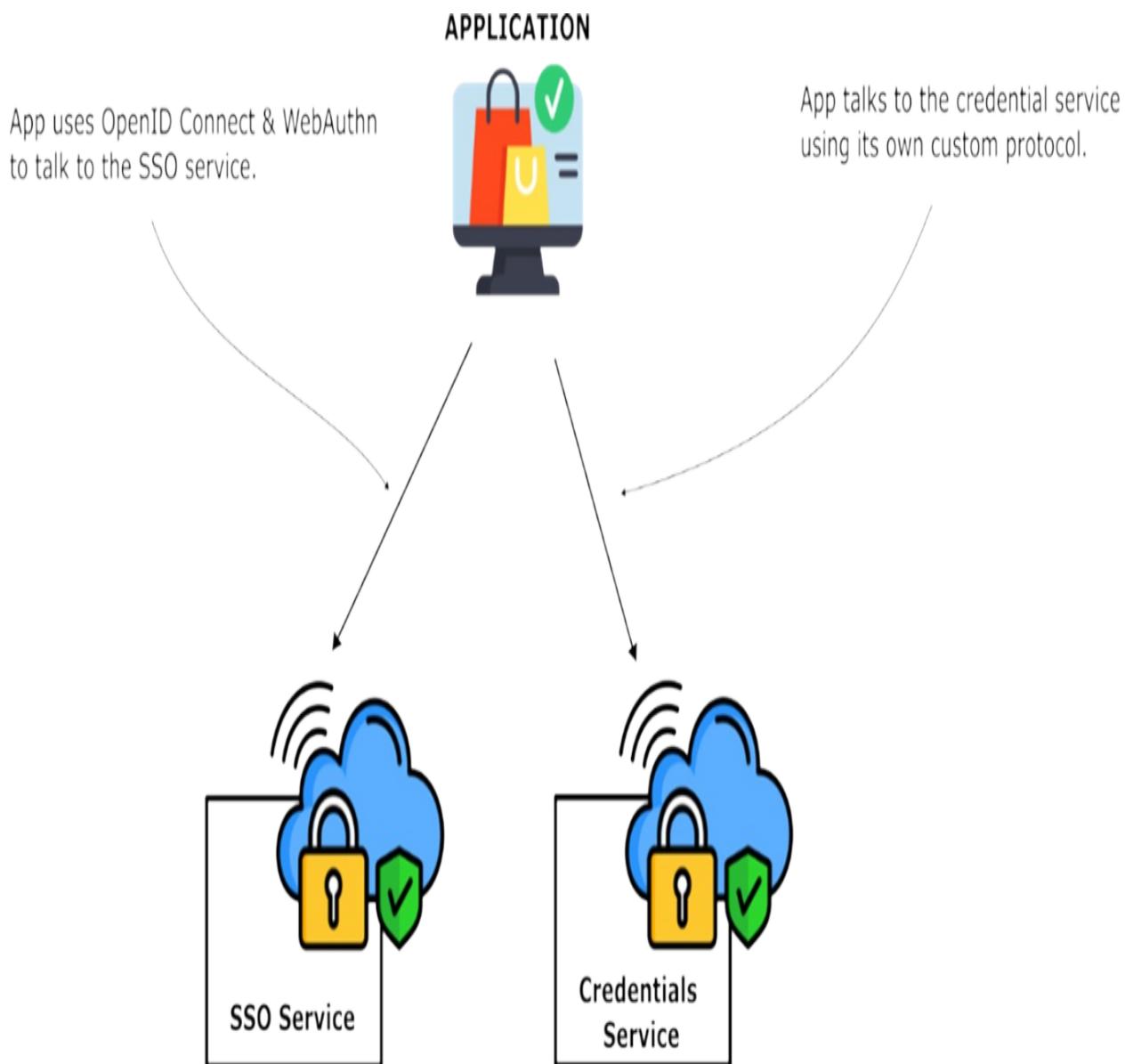
Unfortunately, no industry standard protocol or API for accessing a credential service exists. Every credential service product has its own proprietary API and client libraries that you must add to your application.

Applications access credential services over the network, which means that the credential service must have a way of authenticating the application and sending the request. Thus, we have a bootstrapping problem:

- How does the application obtain the secret to access the credential service?
- Where does the application store the secret to access the credential service?
- How can the secret for accessing the credential service be changed while the application is accessing the credential service?

Bootstrapping trust is a complex problem that can only be solved if baked into the application platform (figure 2.9). Part 3 of the book covers the *Secure Production Identity Framework For Everyone* (SPIFFE), where you will learn how SPIFFE can bootstrap trust for applications running on Kubernetes.

Figure 2.9 A credential management service and a single sign on service are two foundational components for cloud native application security. As a developer you can use the OpenID Connect and WebAuthn protocols to interact with all modern SSO services. However, for credential services there is no industry standard API so you will have to use a proprietary API provided by the credential service implementation.



The techniques for handling secrets securely are the same if you write a

million-line monolithic application or a thousand-line microservice. Getting comfortable with the patterns for handling secrets securely enables you to

- Meet corporate security standard and pass security audits
- Protect credentials for accessing your data sources and APIs
- Simplify automated testing and deployment pipelines

The book teaches the patterns for handling secrets securely and teaches you how to implement them using Spring, Kubernetes, HashiCorp Vault, and public cloud key management services.

BEST PRACTICE

Always store application credentials in a centralized credential store. Acquire a SaaS, commercial, or well-maintained open-source credential management service.

2.2.1 Exercises

8. Where should application secrets be stored?
9. What problems do you run into when you store application secrets in configuration files?
10. What are the benefits of using a credential storage service?
11. Should all applications use a credential storage service?
12. Is there an industry standard protocol or API that can be used to access a credential service?

2.3 Exercise Answers

1. Why are passwords terrible for security and user experience?

Passwords are difficult to manage because long, complex ones are hard to remember. Most users interact with many online services that require passwords, and without a password manager, they often use weak passwords or reuse the same password across multiple accounts.

Additionally, securely storing passwords is both challenging and costly. To improve security and user experience, applications should avoid storing passwords altogether and instead rely on Single Sign-On (SSO)

services.

2. What protocol makes it possible to login without a password?

The WebAuthn protocol enables passwordless login using biometrics, such as a phone's facial recognition, a fingerprint scanner, or phishing-resistant physical security keys.

3. What is OpenID Connect used for?

OpenID Connect (OIDC) is an industry-standard protocol widely implemented by SSO services. It provides a universal API for interacting with SSO services, simplifying the process of authentication and authorization.

4. What is a phishing attack?

A phishing attack tricks users into entering their credentials, such as passwords, into a fake website designed to mimic a legitimate one. This allows hackers to steal sensitive information.

5. Which authentication technology can protect against phishing attacks?

Physical security keys offer protection against phishing attacks by validating the website a user is logging into. The WebAuthn protocol makes it possible to use these keys in applications, ensuring a more secure login process.

6. What capabilities does an SSO service provide?

SSO services handle user authentication on behalf of applications, eliminating the need for the app to store passwords. They enable passwordless login using WebAuthn and provide social login options, such as "Log in with Facebook." Additionally, SSO services can federate with other SSO systems, allowing apps to trust users authenticated by different organizations.

7. Should every application use an SSO service?

Yes, using an SSO service improves security and reduces effort for both developers and users. Even older applications can benefit from being refactored to integrate with SSO services.

8. Where should application secrets be stored?

Application secrets should always be stored in a dedicated credential storage service designed for secure management of sensitive data.

9. What problems do you run into when you store application secrets in configuration files?

Storing secrets in configuration files can lead to several issues:

- They are hard to secure and audit.

- Credential rotation becomes cumbersome.
 - It's challenging to synchronize secrets across multiple machines, increasing the risk of errors and security breaches.
10. What are the benefits of using a credential storage service?
- Credential storage services provide several advantages:
- A single source of truth eliminates configuration drift.
 - Simplifies credential rotation.
 - Offers comprehensive audit logs for better security oversight.
 - Supports hardware security modules for additional protection.
 - Simplifies the implementation of DevSecOps processes.
11. Should all applications use a credential storage service?
- Yes, every application should use a credential storage service because it significantly enhances security and simplifies the implementation of DevSecOps practices.
12. Is there an industry standard protocol or API that can be used to access a credential service?
- Currently, there is no universal industry-standard protocol or API for interacting with credential storage services. Developers must rely on provider-specific APIs and libraries for implementation.

2.4 Summary

- Passwords are weak, reused across services, and expensive to store securely, making them poor for both security and user experience.
- Externalizing authentication to SSO services improves security, reduces development effort, and provides better user experience across multiple applications.
- Modern applications should support OAuth2/OpenID Connect for social login, WebAuthn for biometric authentication, and multifactor password authentication as fallback.
- Corporate SSO services enable employees to access internal applications, external SaaS services, and customer-facing apps using unified Active Directory credentials.
- Organizations can either provide partner employees with internal accounts or delegate authentication to the partner's own SSO service for better security.
- Physical security keys using WebAuthn protocol protect against

phishing attacks by validating website URLs before authentication.

- OAuth2, OpenID Connect, and WebAuthn are the essential authentication protocols developers need to master for modern applications.
- Applications should store sensitive credentials like API keys and passwords in centralized credential services rather than configuration files.
- Credential services provide single source of truth, easy updates, simplified rotation, comprehensive audit logs, and hardware security module support.
- No industry standard exists for credential service APIs, requiring use of provider-specific APIs and libraries.
- Information security engineers choose and configure SSO services, while developers implement OpenID Connect and WebAuthn integration in applications.

[1] <https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html>

3 Service-to-service communication

This chapter covers

- Analyzing problems faced securing service-to-service calls to discover vulnerabilities
- Analyzing technologies available to secure the service-to-service call graph
- Compiling the list of security technologies every developer should know

All applications must solve the following four security problems:

- Securing communication channels
- User authentication
- Handling sensitive credentials such as API keys required to access external services
- Running the application securely in a cloud environment

In a microservice-based application, a single user request can travel between multiple microservices, like a chain of friends trying to pass along a secret. But the whole system falls apart if one friend is a loudmouth or a spy. That's why securing the service-to-service call chain is so important.

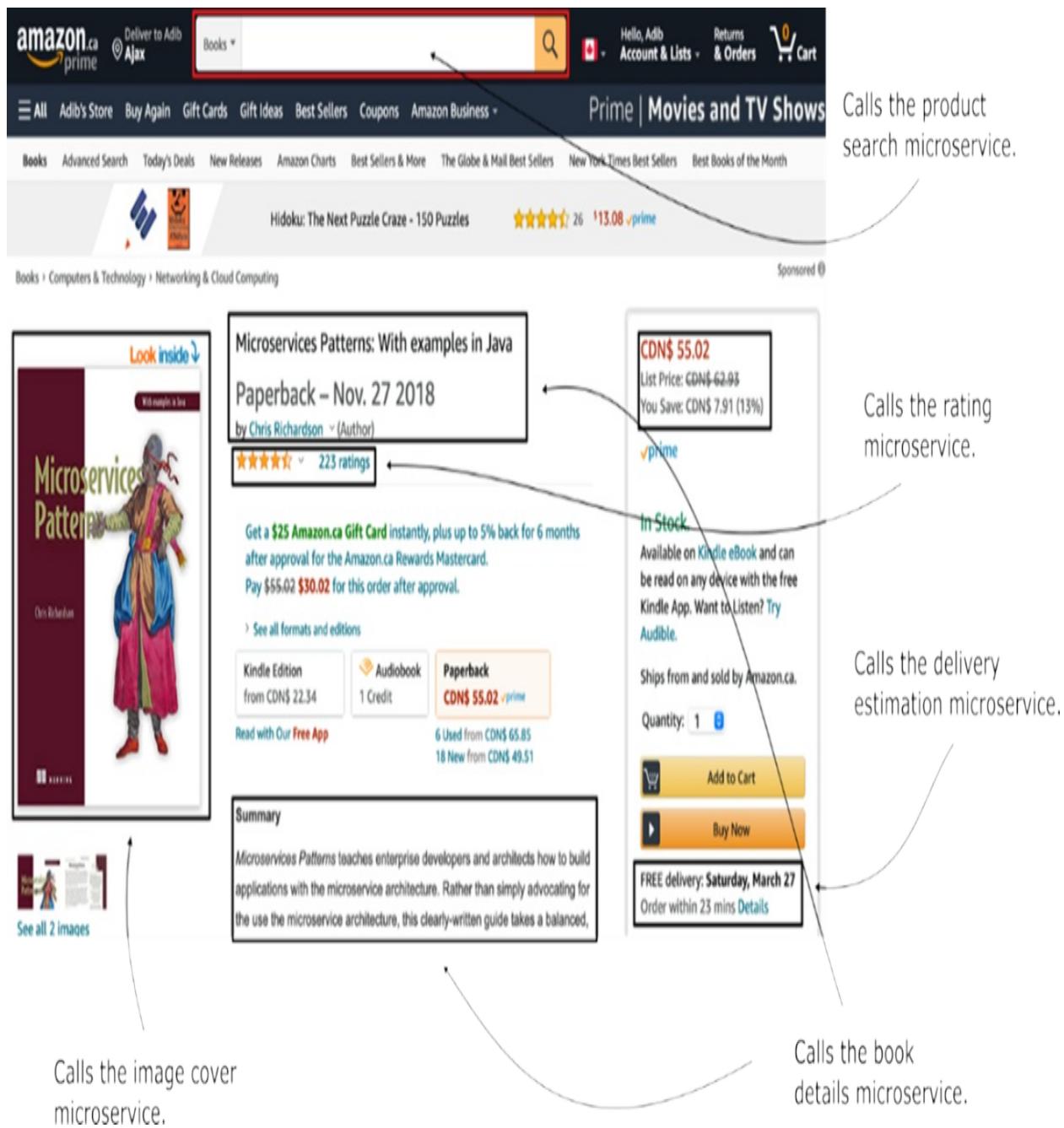
By the end of this chapter, you'll have a solid list of technologies and patterns every developer should know. Like your personal cheat sheet for cloud-native development. With this knowledge, you'll be ready to lock down your app tighter than a teenager's diary. Because in the world of microservices, security isn't just a feature, it's your app's reputation on the line.

3.1 Securing the service-to-service call chain

In a microservice based architecture, an application is decomposed into features organized around business capabilities. Each capability is delivered as an independently deployable microservice. Every microservice has a user interface or API and business logic. If a microservice requires a database,

then the database should be private to the microservice so that microservices can be deployed independently of each other. For example, consider the amazon.com product page in figure 3.1, which highlights potential microservices.

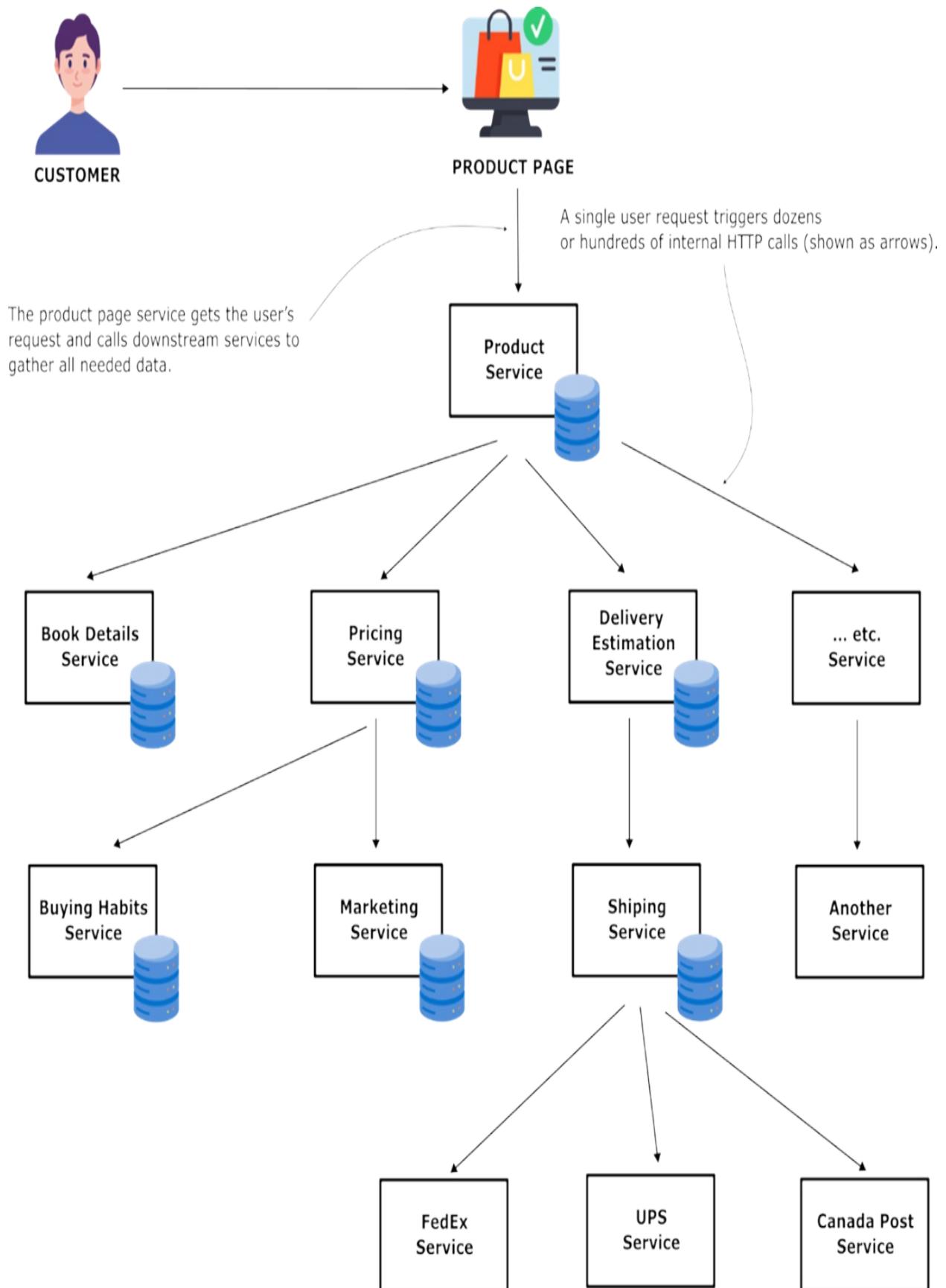
Figure 3.1 Potential microservices that can be easily spotted from a product page: product search microservice, look inside microservice, book details microservice (title, author, publication date, summary), product rating microservice, pricing microservice that can calculate discounts dynamically, delivery estimation microservice that computes when order arrives at a customer's address and shipping costs



The product details page shown in the figure above can be thought off as a microservice. It has a UI that is the HTML and JavaScript required to display the page. It has business logic that makes API calls over HTTP to supporting microservices to get all the data required to render the page. Figure 3.2 shows how the product page microservice is composed out of other microservices.

Figure 3.2 An HTTP request to the product page microservice service fans out through multiple layers of supporting microservices, this is what we mean by the call service-to-service call graph.

Organizations can have hundreds of microservices that interact with each other.



The book details microservice stores facts about the book that don't change, such as the title, author, summary ... etc. The service offers a REST API to get book details using the product ID. There is not much business logic in the book details service mostly data access, and caching logic to make the service is as fast as possible. Since book details don't change once a book is published, the service uses a document database such as MongoDB containing JSON objects to store the book details.

The pricing microservice offers a REST API as its user interface for consuming other services. The business logic dynamically computes the discount offered by factoring in inventory levels, buying habits, market intelligence, and other factors to determine the optimal profit-maximizing price for the item being sold to the person viewing the product. The pricing microservices uses an in-memory data grid such as Apache Geode to cache all the data required to make a pricing decision quickly.

The delivery estimation microservice offers an API to determine how quickly the product can be delivered and the shipping costs, for example “FREE delivery: Saturday March 27 Order within 23 min” as shown in the amazon product page above. To create the delivery, estimate the service does the following:

- Call the inventory service to determine which warehouses have the product in stock.
- Computes the shipping cost and delivery time from each warehouse that has the product in stock.
- Select the optimal warehouse to ship the product from.
- Return the delivery estimate result with details on shipping cost, deliver date, and order cut-off time so that the product page service can display a message such as “FREE delivery: Saturday March 27 Order within 23 min”.

The look-inside service allows customers to flip through a book before purchasing. It uses a JavaScript library to render the book pages and navigate. The business logic ensures that the user can only read a few pages but not the whole book. The book content can be pulled from an object store or document database.

Microservices are both a technical and human organization architecture. Technically, a microservices architecture decomposes functionality into smaller services that call each other, as we have seen in the discussion above. Organizationally, each microservice is owned by a cross-functional team responsible for every aspect of the microservice, including requirements, design, development, testing, deployment, maintenance, and operations. Organizing teams around microservices enables the rapid evolution of features and functionality.

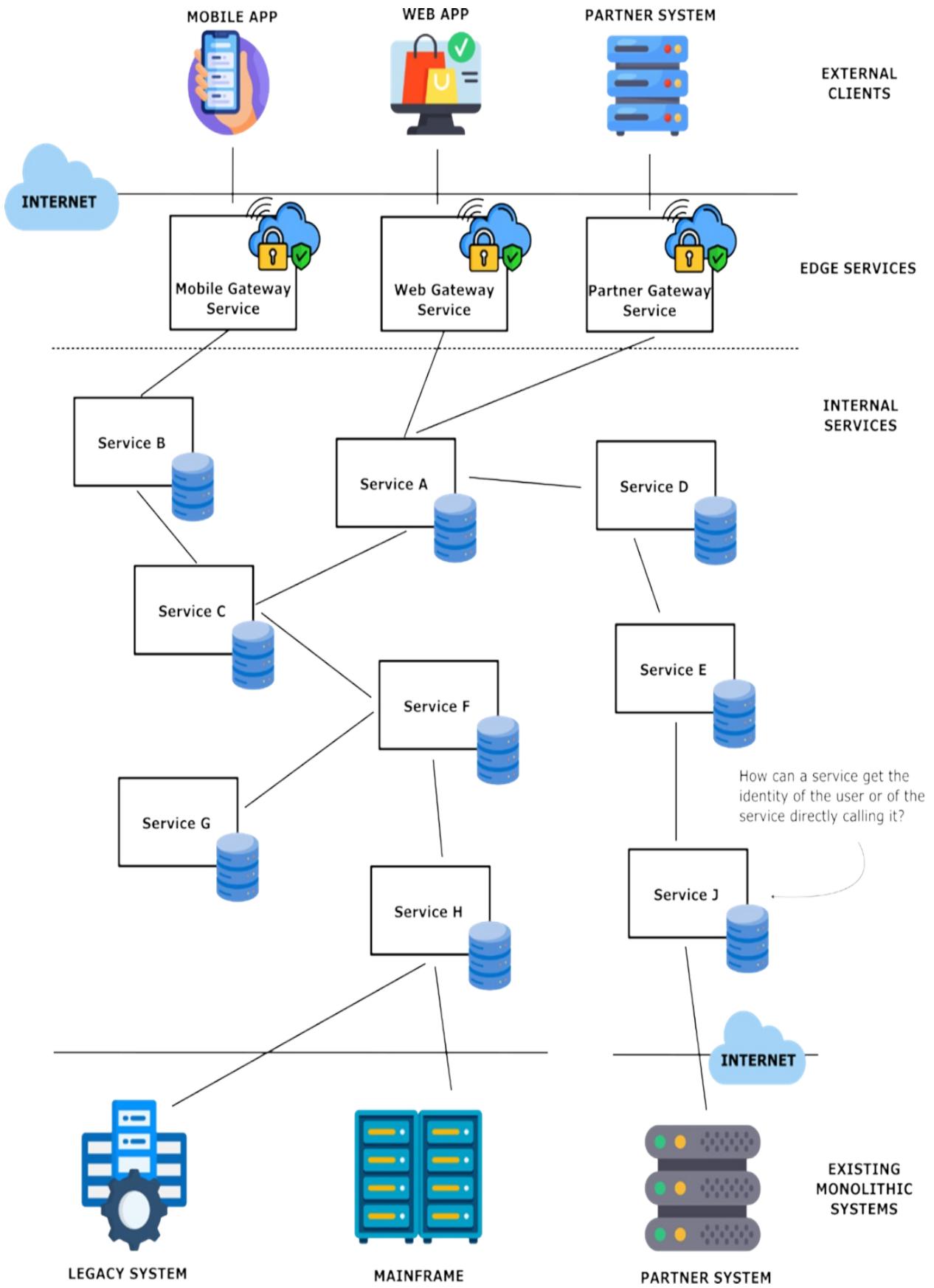
TIP

Microservices increase complexity, and should only be used where appropriate. A lot has been published about microservices over the past few years, covering what microservices are, how to design and build them, and the pros and cons of the approach. If you want to review microservices architecture, check out “Microservices Patterns” by Chris Richardson and “Cloud Native Patterns: Designing change-tolerant software” by Cornelia Davis. The rest of this book assumes basic familiarity with the concept of microservices. You don’t need to be a microservices guru to learn the security aspects.

When one microservice gets a request, it might need to send out several requests to other microservices. This creates two issues. First, how can the user’s identity be passed along from one microservice to another? Second, how can a microservice know which service made the request?

These two problems are illustrated in figure 3.3.

Figure 3.3 A generic call graph showing edge microservices that receive user requests and then call internal microservices to handle the request. How can a microservice determine the user's identity that initiated the request? How can a microservice determine the identity of the service that called it?



3.1.1 Propagating user identity through the service call chain

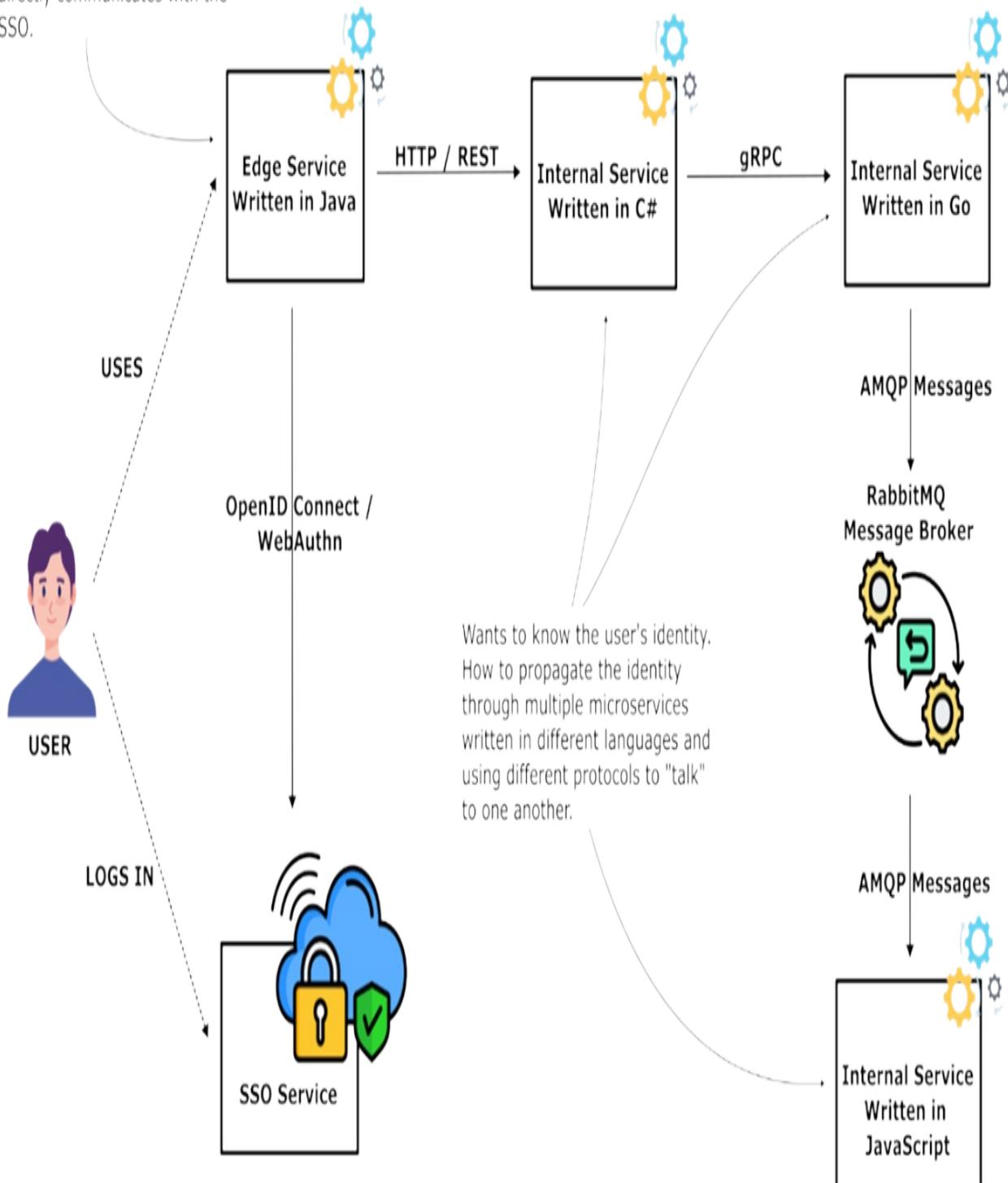
An edge microservice can use OpenID Connect or Web Authentication protocols to determine the identity of the user because it interacts with the user's device. When the edge microservice makes a call to an internal service, it can do so using one of several interaction patterns and protocols:

- Synchronous call using HTTP REST - A straightforward way to request data or perform an operation, where the caller waits for a response over HTTP.
- Synchronous call using an RPC protocol (such as gRPC or Thrift) - A method that uses a remote procedure call protocol for faster communication and stricter type-checking.
- Asynchronous request-reply using a message broker such as RabbitMQ - A way to send a request without waiting immediately, allowing the service to handle the reply later through a message queue.
- Asynchronous event notification using message brokers such as Kafka - A method to broadcast events to multiple services without waiting for a direct response.

The edge service making the call might be written in Java, while the service receiving the call might be written in JavaScript. As shown in figure 3.4, we need a way to propagate user identity through a service-to-service call chain using different network protocols and programming languages.

Figure 3.4 The edge service uses OpenID Connect or WebAuthn protocols to determine the identity of the user making the request. It must then propagate the user identity over HTTP to the service written in C#, which must propagate the user identity over gRPC to the service written in Go, which must propagate the user identity over AMQP to the service written in JavaScript.

Knows the user's identity since it directly communicates with the SSO.



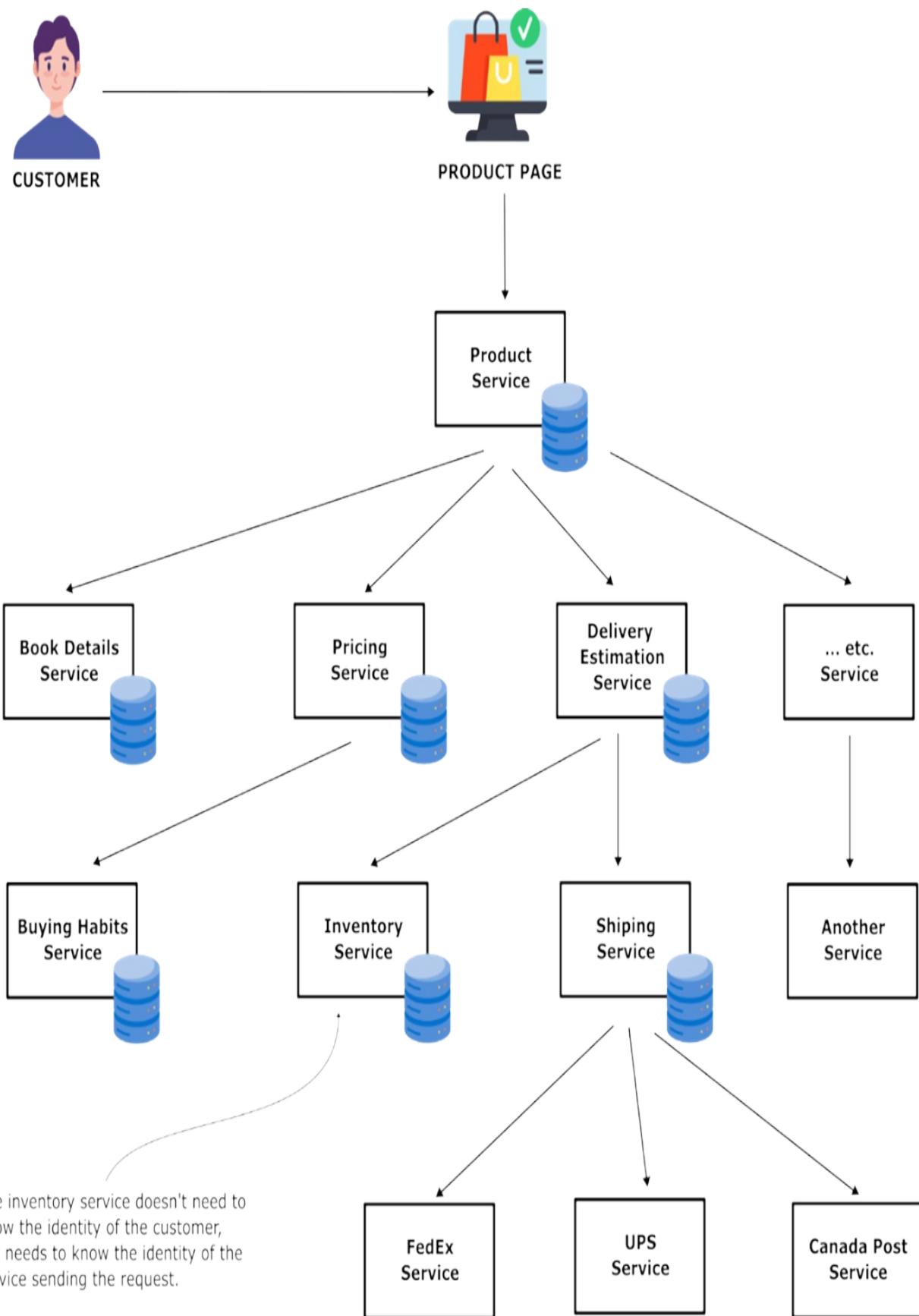
There is no standard way to solve the user identity propagation problem. You must assemble a solution from a set of conventions, patterns, and technologies. We present the patterns in the final part of this book so that you

can build up the pre-requisite knowledge in cryptography primitives, mutual TLS, the JOSE suite of standards, X.509 certificates, OAuth2, OpenID Connect, Service Mesh, and API gateway. Be patient! There is a lot of complex technology to digest and understand before we can solve this problem.

3.1.2 Determining service identity

Some microservices do not care about the identity of the user who initiated the request. For example, the inventory service shown in the diagram below needs the product ID to determine the products' availability. It does not need to know the customer's user ID when browsing the product catalog. Take a look at figure 3.5.

Figure 3.5 Some services, like inventory, do not need to know the identity of the customer who is viewing a product page to return the current inventory level.



The inventory service needs to know the identity of the service requesting so that it can authorize the operation. For example, consider the following operations and associated authorization rules:

- Check product inventory level - any internal service can make a product availability request
- Increase inventory - only the warehouse management service can increase the product inventory count
- Decrease inventory - only the checkout service and the warehouse management service can decrease the inventory count for a product

Microservices make authorization decisions based on user identity, service identity, or both. There are two common approaches to establishing service identity:

- *API keys* - A unique key shared between services to verify their identity.
- Mutual TLS (mTLS) authentication - A secure protocol where both services authenticate each other using digital certificates.

In the API key approach (that we'll talk about in chapter 16), the microservice expects the caller to include an HTTP or message header containing an API key that uniquely identifies the caller. How does the caller get an API key? How is an API key revoked? What is the data format of an API key? Are keys created automatically or manually? API gateways can simplify the implementation of the API key pattern.

In the mutual TLS (mTLS) approach, the caller and the microservice exchange X.509 digital certificates and use them to setup a secure TLS connection (we'll talk about these certificates in detail in chapter 9). The microservice, can determine the identity of the caller by checking the field in the X.509 certificate used to establish the connection.

In chapter 1 we discussed the importance of using TLS to secure all communication channel, mTLS is a stronger configuration of TLS but is more complex to configure and use. Service Mesh and *Secure Production Identity Framework For Everyone* (SPIFFE) simplify the deployment and management of mTLS.

We will explore API gateways and service mesh in the last part of this book so that you can build in your skills in cryptography primitives, mutual TLS, JOSE suite of standards, X.509 certificates, SPFIEE, and OAuth2.

3.1.3 Exercises

1. What is the difference between an edge microservice and an internal microservice?
2. What is the difference between user identity and service identity?
3. Explain the user identity propagation problem?
4. Is there an industry standard solution for solving the user identity propagation problem?
5. What are two approaches used to solve the service identity problem?
6. What technology can help simplify the implementation of API key service identity pattern?
7. What technology can help simplify the implementation of mutual TLS between services?

3.2 Securely running services on Kubernetes

Historically development teams focused exclusively on coding applications. Once development was complete, the application was thrown over the wall to an operations team for deployment and running. Developers did not need to know how to run applications in production, and operators did not need to learn how to write software. The strict division of responsibility between development operations was a constant source of problems for IT organizations. The DevOps movement rose to break down the wall between developers and operations.

Modern software development teams are expected to deploy and run applications in production on self-service public and private cloud platforms. Operations teams have morphed into platform engineering teams that provide developers with a self-service platform to deploy and run applications on.

Kubernetes has emerged as the tool of choice for enterprises to build self-service platforms on. Running applications securely on Kubernetes requires:

- Secure container image
- Secure Kubernetes cluster to deploy the container image into
- Kubernetes deployment manifests that configure the application to run in according with Kubernetes security best practices.

The development team that coded a custom-built application is responsible for containerizing it. Creating an application container is quite easy; there are plenty of examples online. However, many of these examples result in insecure container images. Part 4 of the book shows the best practices for creating secure images using a variety of approaches, including Dockerfiles and Cloud Native Buildpacks.

Installing and securing a Kubernetes cluster is typically the job of a platform engineering team. It is beyond the scope of this book to explain how to deploy, manage, and run a secure Kubernetes cluster.

Running applications on Kubernetes requires developers to write YAML deployment manifests. The deployment manifests can configure every aspect of how an application runs on Kubernetes. Part 4 of the book shows you how to write secure deployment manifests that follow best practices. The book assumes you know the basics of using Kubernetes. “Kubernetes in Action” by Marko Lukšaeo provides an excellent introduction to Kubernetes.

BEST PRACTICE

Learn the best practices for creating secure container images and follow them. Also, learn the best practices for writing secure Kubernetes deployment manifests and follow them.

Roles and Responsibilities

DevOps platform engineers are critical in designing, implementing, and maintaining secure Kubernetes clusters. Their responsibilities include ensuring these clusters adhere to rigorous corporate security standards typically defined and enforced by corporate information security engineers. These standards encompass policies for access control, network security, resource allocation, and monitoring, ensuring that the Kubernetes environment remains robust against potential threats.

As an application developer, your role focuses on securely packaging your application into containers and crafting Kubernetes deployment manifests that follow established security guidelines. This involves addressing vulnerabilities in your application's dependencies, incorporating best practices for container configuration, and ensuring that your application's interaction with Kubernetes resources complies with security policies.

This helps you learn how to securely package your applications and run them on Kubernetes. It covers everything from protecting your container images to writing Kubernetes configuration files that follow security best practices, giving you the knowledge and tools to keep your applications safe and secure.

3.2.1 Exercises

8. What are two skills you should possess as a developer for running services securely on Kubernetes?

3.3 Security technologies every developer should know

The following is a list of the foundational security technologies and standards that every application developer should be familiar with.

- *Transport Layer Security (TLS)*, because you can never trust the network, TLS is the most widely deployed standard to encrypt data communications between applications and the majority of other application security standards depend on TLS.
- *OAuth2 & OpenID Connect (OIDC)*, because you can extract user authentication out of your application and delegate it to a specialized Single Sign On (SSO) service that you can configure quickly and easily. OIDC is very widely supported in the majority of security products and programming language frameworks.
- *Web Authentication (WebAuthn)*, because it can eliminate passwords. You can use it to enable users to login with biometric scans, or phishing resistant physical security keys.

- *Credentials Storage Service*, because modern applications require credentials (passwords, API keys, digital certificates ... etc.) to make calls to external systems. Loosing application credentials leads to catastrophic security failures. Unfortunately, there is no industry standard protocol that works with all products. You will have to learn the generic patterns for securely working with application credentials and the specific details of the credential storage service you are suing such as HashiCorp Vault.
- *API Gateway*, because it simplifies the implementation of API key and can be used to secure edge microservices. The book provides examples, using the Spring Cloud Gateway project.
- Service Mesh, because it simplifies the deployment of mutual TLS between microservices, the book will explain service mesh and provide examples using the Istio service mesh.
- *Secure Production Identity Framework For Everyone (SPIFFE)*, because it offers a good way to solve the problem of bootstrapping trust enabling the implementation of advanced security patterns.
- *Cloud Native Buildpacks*, because it automates the creation of secure container image, that are easy to patch when security vulnerabilities are discovered in base layers.
- *Kubernetes*, because it has emerged as the preferred way to run microservices in production. You will need to know how to write secure Kubernetes deployment manifests to enable security features.

The standards and technologies in the list above depend on the standards and technologies in the list below:

- *Advanced Encryption Standard (AES)*, because AES is the most widely deployed data encryption algorithm. TLS and numerous other standards encrypt data using AES. Also, if you need to encrypt data before storing it on disk or in a database you will need to know AES.
- *JSON Object Signing and Encryption (JOSE)*, because it is a collection of standards that is used by many other standards such as OpenID Connect and numerous products. For example, when you code an OAuth2 based resource server API you will need to know JOSE.
- *X.509 Digital Certificates*, because digital certificates are required by TLS, and they are used for numerous other protocols. If don't know

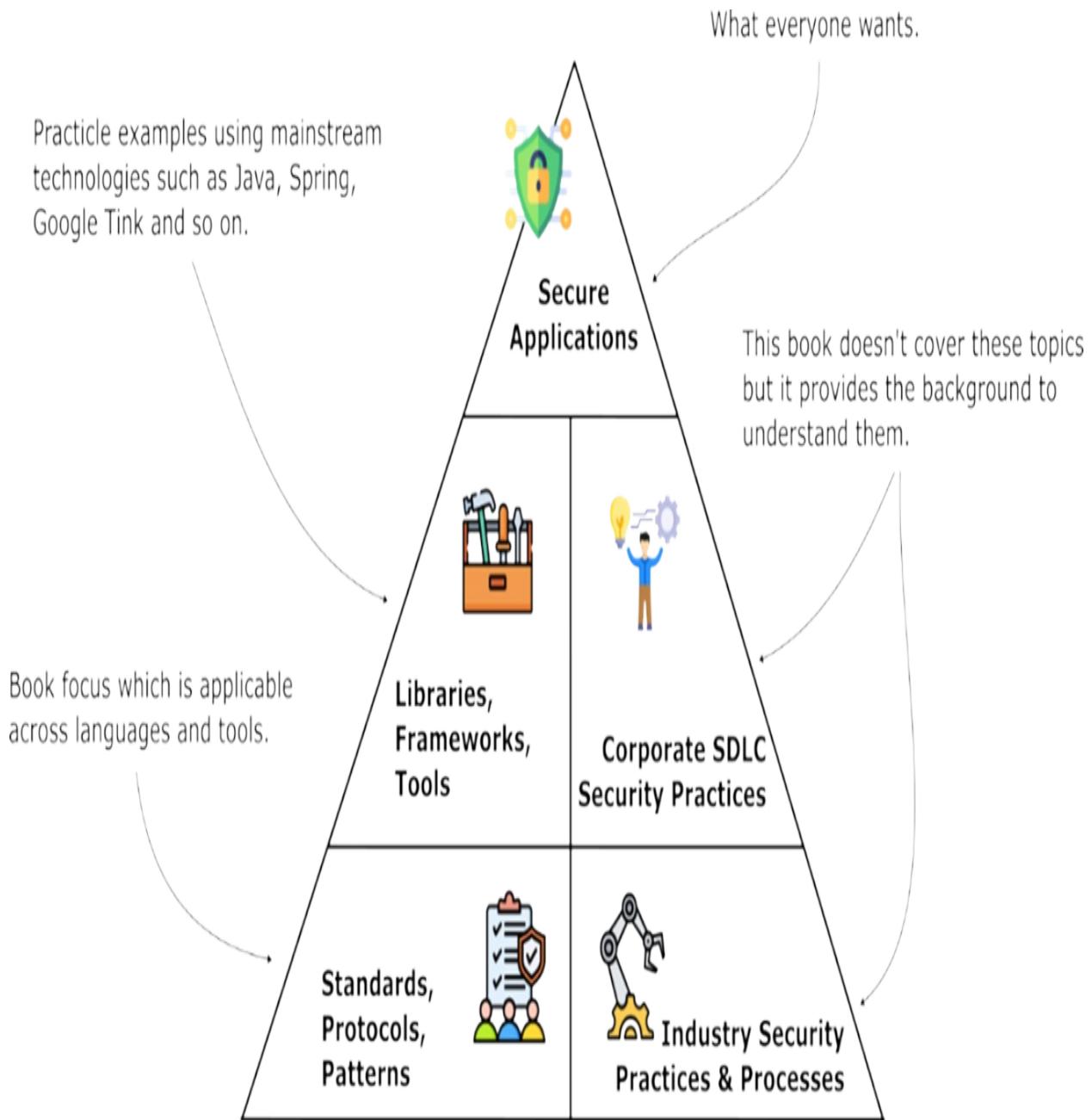
X.509 certificates you will get stuck when reading documentation or debugging issues.

All the previous standards and technologies depend on a solid understanding of the following cryptographic primitives:

- Cryptographic hash functions
- Message authentication codes
- Symmetric key cryptography
- RSA and Elliptic Curve Public key cryptography
- Diffie-Helman key exchange

In part 2 of the book, we will cover cryptography from a developer-friendly perspective. The goal of this book is to teach developers complex security technologies in a developer-friendly way. However, a single book cannot cover everything a developer needs to know about security, so we will stay focused on the areas in the diagram below.

Figure 3.6 The book teaches you the standards, protocols, and patterns for building secure applications using practical sample applications. The goal is to set you up for success on your security learning journey by teaching you foundational technologies every developer should know.



TIP

security is a huge topic, there is a lot of complex technology to learn. You will need to be patient and diligent to master all this technology. If you read the book in the chapter order and run the sample applications, we are confident you will have fun learning all the key technologies for application security that developers should be familiar with.

3.3.1 Exercises

9. Using pen and paper, list all the application technologies a developer should be familiar with, as defined by chapters 1 and 2 of this book.
10. Using a scale from 1 to 10, where 1 means you are a novice and 10 means you are an expert, rate yourself on all the technologies you listen to in the previous question.

3.4 Exercise Answers

1. What is the difference between an edge microservice and an internal microservice?
An edge microservice interacts with users or is exposed over the internet to external systems, it is the entry point for new requests. Internal microservices receive requests from other microservices.
2. What is the difference between user identity and service identity?
User identity is the identity of the entity that initiates a request chain. This is typically a human user such as a customer, employee, or partner. It can also be a system acting on its own behalf, for example API consumer making a request. Service identity is the identity of the service making a request on behalf of a user, for example an edge microservice calling an internal service as part of processing a user request, there are two identities active the user that made the request, the identity of the services in the call chain to fulfil the request.
3. Explain the user identity propagation problem?
Passing user identity between services written in different programming languages using different communication protocols such as HTTP, gRPC, AMQP ... etc.
4. Is there an industry standard solution for solving the user identity propagation problem?
Unfortunately, there is no industry standard way to propagate user identity across service-to-service call. You have to rely on a set of patterns, conventions within your systems.
5. What are two approaches used to solve the service identity problem?
API keys, and mutual TLS are two common approaches to solving the service identity problem.

6. What technology can help simplify the implementation of API key service identity pattern?
API gateway such as Spring Cloud Gateway can simplify the implementation of the API keys pattern.
7. What technology can help simplify the implementation of mutual TLS between services?
Service mesh such as Istio can make mTLS connectivity between microservices simple.
8. What are two skills you should possess as a developer for running services securely on Kubernetes?
Creating secure container images and writing Kubernetes manifests that follow the Kubernetes security best practices.

3.5 Summary

- Microservice architectures decompose applications into independently deployable services that communicate through service-to-service calls, creating complex call chains.
- Edge microservices interact directly with users and external systems, while internal microservices receive requests from other microservices within the system.
- User identity propagation through service call chains requires passing user credentials across different programming languages and network protocols like HTTP, gRPC, and AMQP.
- No industry standard exists for propagating user identity between services, requiring custom solutions using patterns and conventions.
- Service identity can be established through API keys or mutual TLS authentication to authorize which services can perform specific operations.
- API gateways simplify implementation of API key patterns for service authentication and authorization.
- Service mesh technologies like Istio simplify deployment and management of mutual TLS between microservices.
- Developers must learn to create secure container images and write Kubernetes deployment manifests following security best practices.
- Platform engineering teams are responsible for providing secure Kubernetes clusters, while developers handle secure application

packaging and deployment configuration.

- Essential security technologies for developers include TLS, OAuth2/OpenID Connect, WebAuthn, credential storage services, API gateways, service mesh, SPIFFE, Cloud Native Buildpacks, and Kubernetes.
- These technologies depend on foundational knowledge of AES encryption, JOSE standards, X.509 digital certificates, and cryptographic primitives like hash functions and public key cryptography.
- DevOps has evolved from separated development and operations teams to integrated teams responsible for both building and running applications in production.

Part 2: Cryptography foundations

Part 1 gave you the big picture of application security — who does what, where to focus your learning, and how all the pieces fit together. Now it's time to zoom in on the foundation: the cryptographic algorithms themselves. These are the building blocks that make everything else in security possible.

Most developers encounter cryptography when a library throws a confusing error or when they're told to "just add encryption." The math can look intimidating, and the standards read like they're written for rocket scientists. The goal of Part 2 is to cut through that complexity. You'll learn just enough to practically understand the cryptography foundation.

We'll start with the basics: how to guarantee integrity and authenticity with hashes and HMACs (chapter 4). Then we'll move into encryption with AES (chapter 5), learning how to protect confidentiality without falling into common traps. From there, you'll explore public key cryptography with RSA (chapter 6) and elliptic curves (chapter 7), seeing how they solve the key distribution problem and enable digital signatures.

Along the way, the Acme Inc. case study will keep us grounded in real-world scenarios, with Java examples that you can run and adapt. By the end of Part 2, you'll not only understand the essential algorithms behind modern security protocols, but you'll also have the confidence to configure and use them correctly in your own systems.

4 Message Integrity and Authentication

This chapter covers

- Guaranteeing data integrity using the Secure Hash Algorithm (SHA)
- Ensuring sender authenticity using a Hashed Message Authentication Code (HMAC)
- Ensuring data integrity using an HMAC
- Using the Java Cryptography Architecture (JCA) and Extensions (JCE)

This chapter is the first step in a friendly introduction to cryptographic algorithms for application developers. We will not cover the mathematics of the cryptography algorithms. Instead, we will demonstrate cryptography concepts with working Java examples so you can build the intuition and background to understand application security.

Cryptographic algorithms are the foundational security building blocks, no matter what programming language you write code in or which cloud provider you deploy your application on. Terse documentation and mysterious error messages from security libraries make perfect sense if you understand the basics of cryptography. No more getting stuck and blindly copying and pasting from stackoverflow.com and blog posts.

Definition

A cryptographic algorithm is a set of mathematical rules used to keep information secure — by scrambling data so only the right people can read it, or by proving data is genuine and unchanged.

Suppose you want to add a social login button to your application such as “Login with Google”. You will need to use some protocol like the OpenID Connect and OAuth2. OpenID Connect is built on top of the *JSON Web Token* (JWT) standard which means you will need to understand *JSON Web*

Signature (JWS) and *JSON Web Encryption* (JWE). To understand JWS, you must understand Hashed Message Authentication Code (HMAC). To understand HMAC, you need to understand the concept of cryptographic hash functions and Message Authentication Code (MAC). All these names might now sound foreign, but this is exactly what you'll study in this book. I have prepared a step-by-step approach where we start with the needed basics and build from ground up what you need to know.

We will work on two example applications. The first application will teach you how to use a cryptographic hash function to detect data corruption. The second application will teach you how to use a cryptographic function called Hashed Message Authentication Code (HMAC) to determine who created a file and prove that the file was not tampered with.

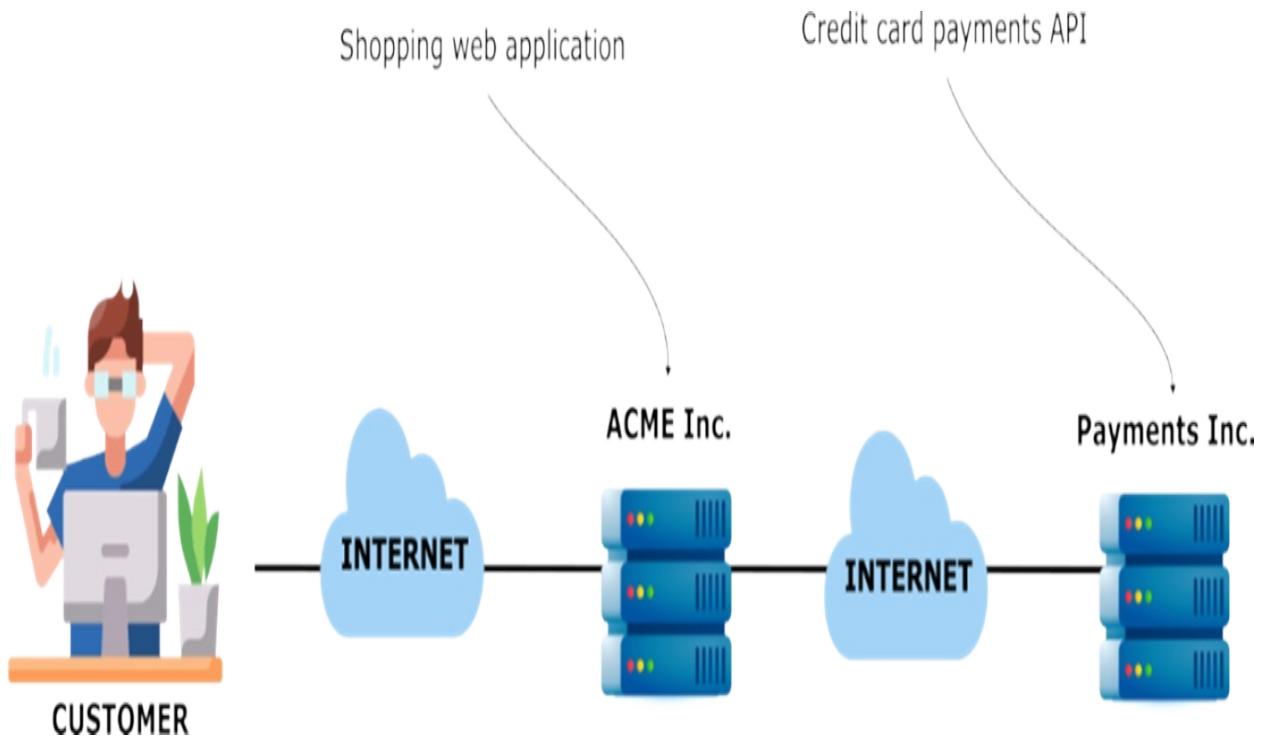
Definition

A cryptographic function is a special kind of mathematical operation used in security to transform data in a way that is hard to reverse without a secret (like a key). Examples include hashing functions, encryption functions, and digital signature functions.

4.1 The goals of cryptography

Consider ACME Inc. a shoe manufacturer. Customers shop for shoes on ACME's e-commerce website and pay for orders using a credit card. For every order placed ACME's e-commerce system calls the credit card API offered by a payment processing company (figure 4.1).

Figure 4.1 A typical e-commerce high level architecture. Customers place orders using the ACME Inc. web applications, which calls the Payments Inc. credit card API to collect payments from customers while processing an order.



Cryptography is used to secure the HTTP requests from the user's web browser to the e-commerce website and from the e-commerce system to the payments API. As HTTP request/response messages flow back and forth between customers, ACME Inc. and Payment Inc. cryptography is used to solve four fundamental security problems:

- *Integrity* Ensures that the data sent in the messages hasn't been altered during transmission.
- *Authentication* Confirms that the parties communicating are who they claim to be.
- *Confidentiality* Protects the data from being read by unauthorized parties, ensuring privacy.
- *Non-repudiation* Guarantees that the sender cannot deny sending the message or transaction.

Integrity ensures that data is not tampered with during transmission or storage. An HTTP request from a customer's browser to the e-commerce system can be altered due to issues in the network hardware or a hacker intercepting and modifying the request. Similarly, a customer address saved to a database can be altered accidentally due to disk drive corruption or intentionally by a hacker modifying the database. One of the fundamental

problems that cryptography solves is ensuring message integrity during transmission or storage.

Authentication in the context of cryptography means that the message's recipient can determine who sent the message. For example, how can the payment API be sure that a request to charge a credit card came from ACME Inc. and not a hacker pretending to be ACME Inc. How can the customer be sure that they are sending their credit card information to ACME Inc's website rather than a hacker pretending to be ACME's website?

Note

Authentication can refer to user authentication or message authentication. User authentication typically means asking users to prove their identity via a challenge, such as providing a correct username/password combination. In this part of the book, when you see the term authentication, we mean message authentication as described above rather than logging users into an application.

Confidentiality ensures that data is only understandable by the intended recipients. For example, how can ACME Inc. and Payments Inc. ensure that hackers cannot steal credit card details if they can intercept network communications? If an employee of ACME Inc. loses their laptop with customer data on it, how can a customer be sure that their personal details are not accessible to a random stranger who finds the laptop?

Non-repudiation is a legal concept, it means that one party in a transaction cannot deny having done something. For example, how can ACME Inc. prove to a court of law that the payment API received and approved a request to charge \$100 to a customer's credit card. Authentication allows two parties Alice and Bob to establish each other's identity, but a third-party Judy can't establish their identity. Non-repudiation allows Bob to prove to a neutral third party, Judy, that he got a message from Alice and allows Alice to prove to Judy that she got a message from Bob. Non-repudiation is critical for establishing legal validity of interactions between computer systems.

Considering security along the dimensions of integrity, authentication, confidentiality, and non-repudiation provides a framework for understanding

how and when to use the various cryptographic algorithms. The following table shows the various types of algorithms required to achieve the goals of cryptography (table 4.1).

Table 4.1 Cryptography Goals and Algorithm Types

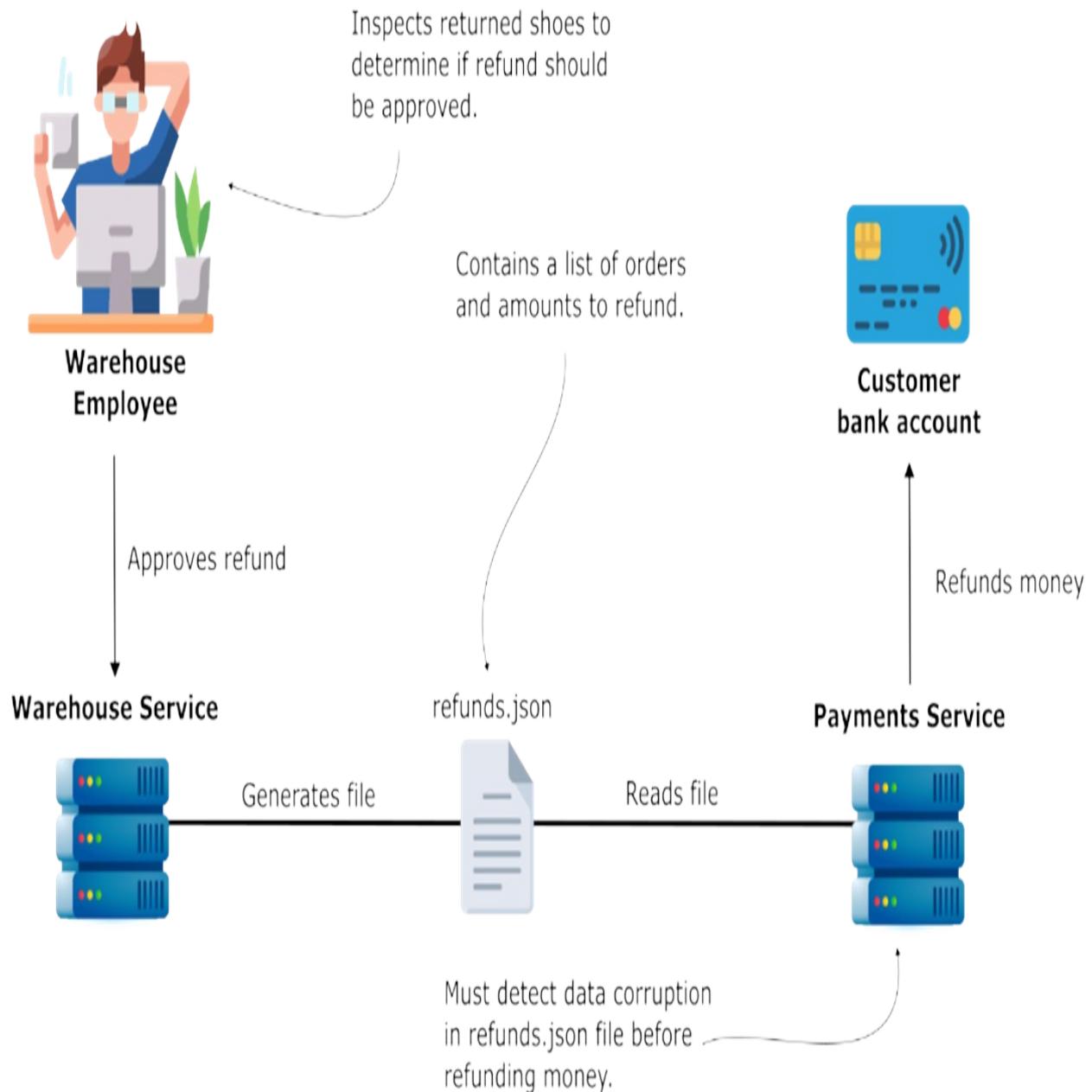
Goal	Foundational algorithm required	Standards Covered in Book
Integrity	Cryptographic hash function	SHA-2, SHA-3 (this chapter)
Authentication	Cryptographic hash function	HMAC using SHA-2 or SHA-3 (this chapter)
Confidentiality	Symmetric or public key encryption	AES, RSA, ECC, JWE, Diffie-Helman, TLS
Non-repudiation	Public key encryption	RSA, ECC, X.509, JWS, PKI

Learning the algorithms in the previous table will take several chapters. It is a lot of effort, but it is totally worth it, as it will give you programming superpowers. Grab a coffee and your laptop. From here on, we will teach you how to guarantee data integrity and authentication.

4.2 Cryptographic hash functions

ACME Inc., an online shoe retailer, allows customers to return shoes they don't like for a full refund. Customers mail the returns to ACME Inc's warehouse, where staff checks that the returned shoes are in good condition before authorizing a refund. Once per day, the warehouse management application generates a `refunds.json` file containing a list of orders and the amount to refund. The payment service issue refunds to customer credit cards based on the data from the `refunds.json`. Figure 4.2 illustrates this workflow.

Figure 4.2 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a `refunds.json` file. The payment service refunds customer credit cards for the amount specified in the `refunds.json` file.



Next snippet shows an example of `refunds.json` file content.

```
[
  {
    "orderId" : "12345",
    "amount" : 500
  },
  {
    "orderId" : "56789",
    "amount" : 250
  }
]
```

The business wants to ensure customers happiness. Therefore, the payment service must return the correct amount of money to the correct customer credit card. To accomplish the business goal, the payment service must be able to detect data corruption in the `refunds.json` file before it starts processing refunds. This section explore how cryptographic hash functions can be used to detect data corruption.

NOTE

The rest of the book uses the ACME Inc. scenario outlined earlier. The book provides a set of sample applications that implement variations of the ACME Inc. scenario using a variety of cryptographic algorithms. All the code for the sample application can found at <>insert link>>. Using the same scenario in multiple chapters will make the concepts in the book easier to understand. You will see the diagram above repeated where needed in the book, so you don't have to flip back to this section.

4.2.1 Secure Hash Algorithm (SHA)

A *hash function* takes an input of any size and maps it to a fixed size bit string. For example, executing the SHA-256 hash function on the string “abc” or a 4GB movie file will produce a 256-bit output string. Table 4.2 shows example inputs and outputs from the SHA-256 hash function.

Table 4.2 Hash values using SHA-256 for inputs of varying sizes

Input	SHA-256 hash value represented as hexadecimal number
1-byte lowercase “a”	ca978112ca1bbdcfac231b39a23dc4da786eff8147c4e72b980771
1-byte uppercase “A”	559aead08264d5795d3909718cdd05abd49572e84fe55590eef31c
2.6GB ubuntu.iso file	b45165ed3cd437b9ffad02a2aad22a4ddc69162470e2622982889c

Hash functions are deterministic. This means that every time a hash function is executed on the same input, it produces the same output value.

Cryptographic hash functions are a special type of hash function with mathematical properties that make them suitable for computer security. Two primary properties of a cryptographic hash function are:

- *One-way property*: Given the hash function output, it is not feasible to determine the original input that the hash function executed on (figure 4.3).
- *Collision resistance*: different input values produce completely different output values. Finding two inputs that produce the same hash value should not be possible.

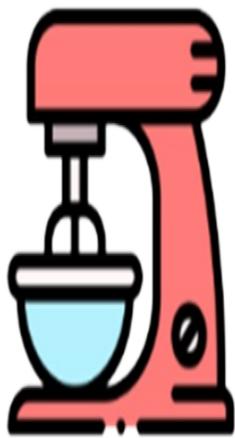
Figure 4.3 Think of the hash function as a blender that takes inputs of different sizes to produce outputs of the same size. Different input values produce completely different output values. Attackers with access to the hash value cannot determine anything about the input.



**Input of any number
of bytes**



The same input always produces the same output
If the input varies even by 1 bit, the output will
be completely different.



Someone in possesion of the output value
cannot determine the input value.



**Output of fixed lenght
(hash value)**



Much mathematics is used to design and assess the security of cryptographic hash functions. It takes years of effort, much scrutiny, and peer review by cryptographers to reach a consensus that a particular hash function is safe for use in cryptography. You should always use standardized peer-reviewed hash functions.

TIP

Designing and implementing a secure hash function is a huge undertaking full of pitfalls. Do not design your own cryptographic hash function. Use only industry-standard functions that have been peer reviewed and approved by your corporate information security team. All programming languages have excellent implementations of cryptographic hash functions as part of the standard libraries, so there is no need to implement your own hash function.

The Secure Hash Algorithm (SHA) is family of widely used cryptographic hash functions standardized by the National Institute of Standards and Technology (NIST). NIST is a USA federal government body that defines the cryptography standards for use in American government applications. The American government is a massive technology buyer. Companies selling software to the American government must adhere to the standards produced by NIST, so NIST standards are widely implemented in all programming languages. There are four generations of SHA algorithms:

- SHA-0 published in 1993, withdrawn due to security issues
- SHA-1 published in 1995, deprecated in 2011 due to security weaknesses
- SHA-2 published in 2001, widely used at time of writing and still considered secure
- SHA-3 published in 2015, being adopted in new applications and standards

The previous list illustrates that algorithms can become insecure over time. Cryptographers are always trying to find weaknesses in widely used hash functions because breaking a hash function breaks all protocols built on top of it. For example, Google[\[1\]](#) security researchers found a way to generate collisions with the widely used SHA-1 hash function. As a result, SHA-1 is

no longer considered secure.

TIP

Today's secure algorithms might be considered insecure tomorrow. You should always use an up-to-date industry standard hash function that is considered secure by the cryptography community and your corporate information security team. Always be ready to change your code in response to new attacks.

Because it takes a long time to standardize a hash function, NIST thought it was prudent to have a backup hash function in case a successful attack against SHA-2 was discovered. The SHA-3 hash function uses a different mathematical structure to SHA-2, so a mathematical breakthrough affecting SHA-2 should not affect SHA-3.

The SHA-2 and SHA-3 algorithms can be configured to produce 224, 256, 384, or 512 bits of output. The longer the output the more security you get. Selecting the output size to use depends on a variety of memory, speed, and security trade-offs. Today, 256-bit is the minimum setting that is considered secure. Research the recommended output size before you write code using a cryptographic hash function. For example, www.keylength.com aggregates recommendations from various government organizations on the minimum key sizes that should be used for various cryptographic algorithms.

TIP

If you have implemented a `hashCode()` method on a Java object, you might wonder how the Java `hashCode()` method is related to cryptographic hash functions. The `hashCode()` defined on a Java object returns a 32-bit integer which is insufficient for cryptographic use. NEVER! use the Java `hashCode()` method for cryptography.

4.2.2 Verifying integrity using a cryptographic hash function

Because cryptographic hash functions produce a unique output for each unique input, they are ideal for detecting data corruption. In the ACME Inc.

Scenario discussed earlier, the warehouse management application produces two files:

- `refunds.json` contains the orders ids and refund amounts
- `refunds.json.sha256` contains the value of the SHA-256 function computed over the contents of the `refunds.json` file

The payment service detects data corruption using the following steps:

1. Compute the SHA-256 hash value on the contents of the `refunds.json` file.
2. Compare the computed SHA-256 value to the expected value found in the `refunds.json.sha256` file. If the values match there was no data corruption. If the values don't match then `refunds.json` or `refunds.json.sha256` files are corrupt, an error should be raised and no credit cards should be refunded.

Take a look at table 4.3, which contains a pseudo-code example of our two apps which are part of our example.

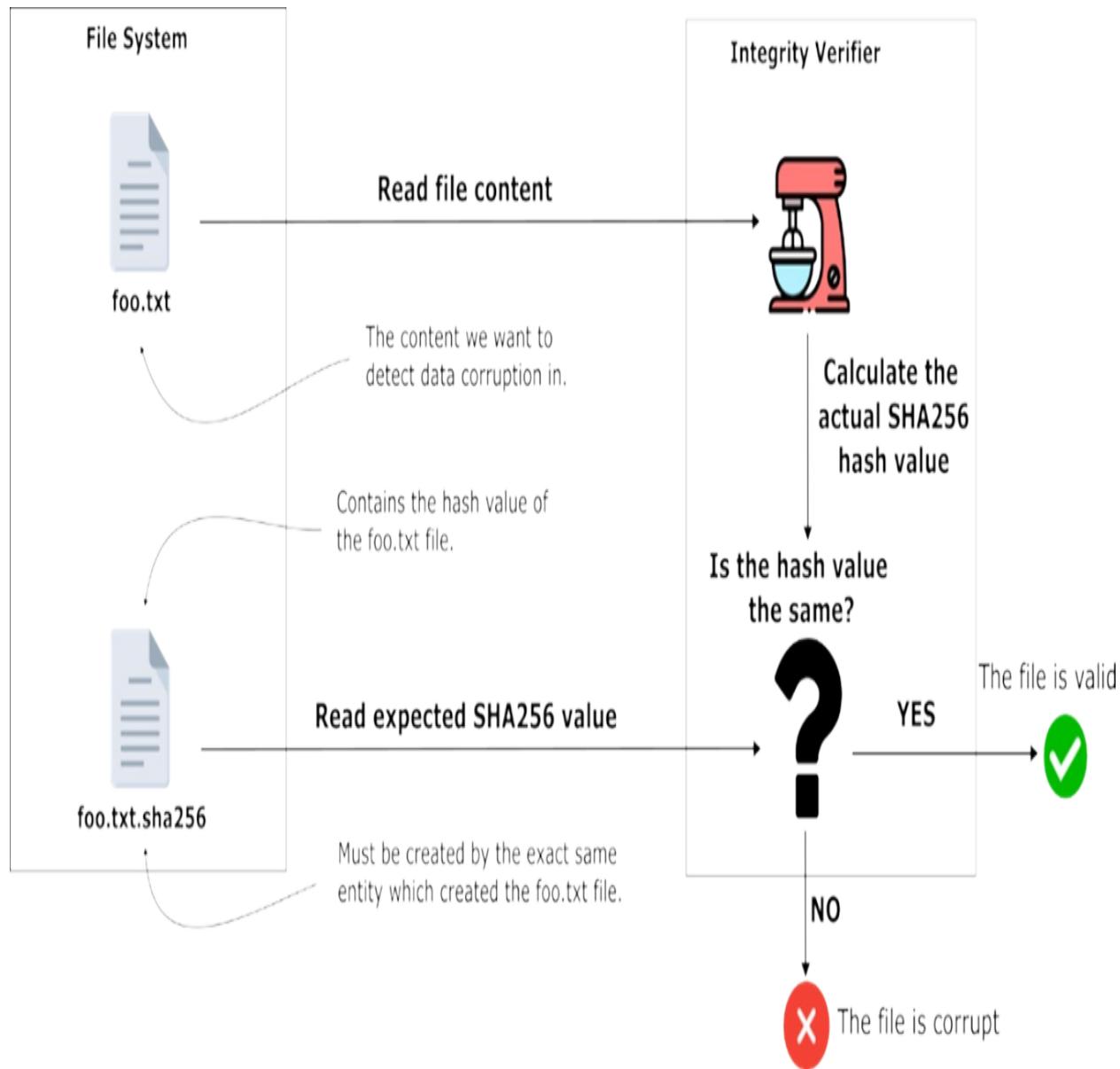
Table 4.3 Pseudo code for using a cryptographic hash function to validate file integrity

Warehouse Management Application	Payments Service
<pre>saveRefunds(Path location, bytes[] content) hashValue = CryptoUtils.sha256(content) writeFile(location, content) writeFile(location + ".sha256", hashValue)</pre>	<pre>isRefundsFileValid(Path location) content = readFile(location) expected = CryptoUtils.sha256(content) actual = readFile(location + ".sha256") if(expected == actual) return true else return false</pre>

Figure 4.4 shows the generic validation process graphically.

Figure 4.4 Detecting data integrity violation using a cryptographic hash function. The `foo.txt` producer computes the hash value of the content and then writes it to `foo.txt.sha256`. A consumer computes the SHA-256 value of the contents of `foo.txt` and then compares the computed value to the expected value in `foo.txt.sha256`. If expected and actual values match, the file is valid; if not,

the file is corrupt.



You might be wondering what happens if the `foo.txt` file is valid but `foo.txt.sha256` is corrupt. In such a case, the verification algorithm will reject a valid `foo.txt` file, thinking it is corrupt. This behavior is ok because our goal is to process the `foo.txt` file if and only if we are 100% sure that it is valid. It is better to reject some valid files than process an invalid one.

A hacker can edit the `foo.txt` file, compute a new SHA-256 value, and write it to `foo.txt.sha256`. Will the verification algorithm catch active

tampering? The answer is no. There is no way for the verification algorithm to protect against an attacker tampering with both `foo.txt` and `foo.txt.sha256`. The algorithm is designed to protect against accidental data corruption caused by network errors, disk failure, and accidental editing of the files. We will discuss how to protect against active tampering later on. Let's examine how maven central and git use cryptographic hash functions.

Maven central and Cryptographic hash functions

Maven Central is the Java community's repository for distributing open-source libraries. It contains millions of files downloaded millions of times per day. Data corruption due to network errors or disk failure will occur. Maven Central uses cryptographic hash functions to enable tools to detect data corruption. Figure 4.5 shows the Spring Framework's Web MVC module files stored on the Maven Central repository.

Figure 4.5 Files on maven central. Notice the sha1 and md5 files that contain hashed values of the corresponding artifacts. MD5 and SHA-1 are insecure, but they are still used by maven central for backward computability reasons.

org/springframework/spring-webmvc/6.2.10

File and its associated SHA-1 hash value.

.. /

spring-webmvc-6.2.10-javadoc.jar	2025-08-14 07:37	2598426
spring-webmvc-6.2.10-javadoc.jar.asc	2025-08-14 07:37	833
spring-webmvc-6.2.10-javadoc.jar.md5	2025-08-14 07:37	32
spring-webmvc-6.2.10-javadoc.jar.sha1	2025-08-14 07:37	40
spring-webmvc-6.2.10-sources.jar	2025-08-14 07:37	854135
spring-webmvc-6.2.10-sources.jar.asc	2025-08-14 07:37	833
spring-webmvc-6.2.10-sources.jar.md5	2025-08-14 07:37	32
spring-webmvc-6.2.10-sources.jar.sha1	2025-08-14 07:37	40
spring-webmvc-6.2.10.jar	2025-08-14 07:37	1091514
spring-webmvc-6.2.10.jar.asc	2025-08-14 07:37	833
spring-webmvc-6.2.10.jar.md5	2025-08-14 07:37	32
spring-webmvc-6.2.10.jar.sha1	2025-08-14 07:37	40
spring-webmvc-6.2.10.module	2025-08-14 07:37	4206
spring-webmvc-6.2.10.module.asc	2025-08-14 07:37	833
spring-webmvc-6.2.10.module.md5	2025-08-14 07:37	32
spring-webmvc-6.2.10.module.sha1	2025-08-14 07:37	40
spring-webmvc-6.2.10.pom	2025-08-14 07:37	2963
spring-webmvc-6.2.10.pom.asc	2025-08-14 07:37	833
spring-webmvc-6.2.10.pom.md5	2025-08-14 07:37	32
spring-webmvc-6.2.10.pom.sha1	2025-08-14 07:37	40

When the Maven Central service was first released, the SHA-1 and MD5 cryptographic hash functions were considered secure and industry standard. You can see in the screenshot a set of files that end in md5 and sha1 extensions for each of the files stored in Maven Central. Now, both SHA-1 and MD5 are considered to be insecure. Maven central still needs to support SHA-1 and MD5 for backward compatibility reasons. There is an open ticket[\[2\]](#) to add support for SHA-256 and SHA-512 to Maven Central.

GIT and Cryptographic hash functions

The Git source control system uses a cryptographic hash function to identify every file and every commit in the repository. The `git log` command displays a list of all commits in a repo, including the SHA-1 hash of all commit contents. The next snippet is an example of a Git commit identified by the SHA-1 of the commit's content.

```
commit 0303595cb21647b203ae9069a5191cbd4ad0f865 #A
Author: Adib Saikali <adib@example.com>
Date:   Sun Aug 23 22:05:34 2020 -0400
```

Project Skeleton

Git identifies every file in the repository using the SHA-1 of the file. Git performs the following steps when you add a file to the repository:

1. Computes the SHA-1 of the file being added.
2. Checks to see if the SHA-1 value already exists in the repository.
 - a. If the SHA-1 exists in the repo, Git adds an entry in its database where the key is the file path, and the value is the SHA-1 of the file. Since the content is already in the repo it does not need to add it.
 - b. If the SHA-1 is not in the repo. Git adds the file content to its database with a key set to the SHA-1 hash. It then adds an entry linking the path name of the file to the SHA-1 key.

The SHA-1 function allows git to store a single instance of a file in the repo, even if the file exists in multiple directories. Without cryptographic hash functions, source control systems like git are impossible to implement.

Git was designed in 2005, and its code and data structure were hard-coded to use SHA-1. SHA-1 was deprecated in 2011. In 2018 the git team selected SHA-256 to replace SHA-1. The change from SHA-1 to SHA-256 is a massive undertaking, and the git team has been working on the upgrade for several years[\[3\]](#).

4.2.3 Design for hash function change

The Maven Central and Git examples discussed in the previous section demonstrate the need for applications to upgrade their hash functions. If you are designing a system that uses a cryptographic hash function, assume that it is only a matter of time before you change it.

Design your code, data structures, file formats, and database schemas so you can change the hash function you use. For example, you can include a version number in your data format to allow consumers to determine which hash algorithm was used.

WARNING

Attackers will set version fields to known to old values to weaken the security of a data exchange. Applications that read the version field should be able to upgrade old data from an insecure algorithm version to a more secure version. Applications should take special care to ensure that it is impossible to downgrade from a secure version to an insecure one of an algorithm. Any versioning scheme must go through a proper security analysis.

Cryptographic hash functions are super useful for various applications. They are used in all the security protocols you need to know as a developer, including AES, TLS, JWT, JWE, JWS ... etc. Getting comfortable with cryptographic hash functions adds a superpower to your programming toolbox. Cryptographic hash functions can help you achieve the cryptographic objectives outlined in Table 4.4.

Table 4.4 Cryptography goals and algorithms matrix

Goal	SHA-2	SHA-3
Integrity	Yes	Yes
Authentication	No	No
Confidentiality	No	No
Non-repudiation	No	No

All programming languages have excellent implementations of cryptographic hash functions as part of the standard libraries. The samples in this book are written in Java. The next section provides an overview of the Java

Cryptography Architecture and Extensions, which we will be using in many samples throughout the book.

4.2.4 Exercises

1. What are the four main goals of cryptography?
2. How is a cryptographic hash function different from the hashCode() method in Java?
3. What problem does using SHA-256 on a file solve, and what problem does it not solve?

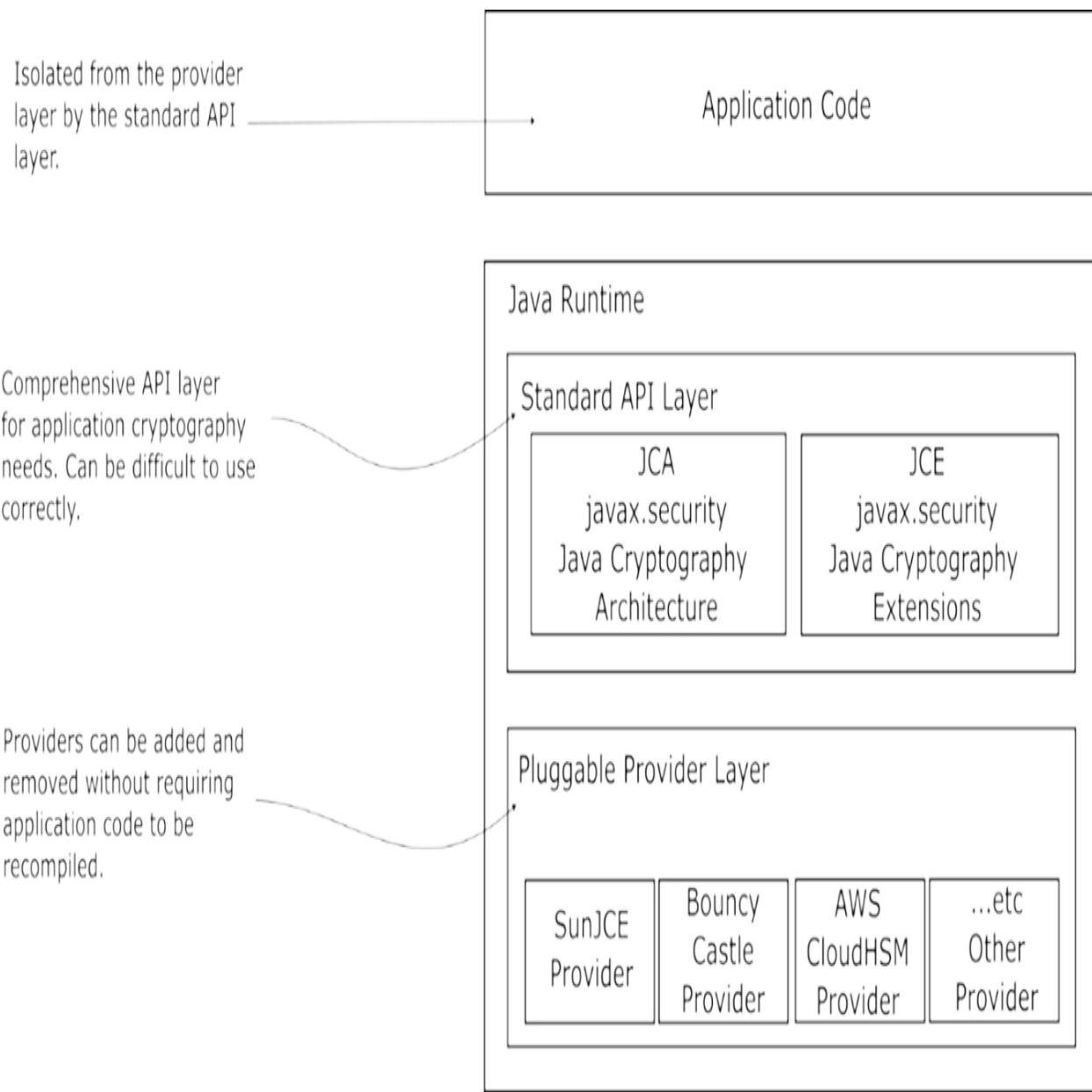
4.3 Java cryptography architecture and extensions

The *Java Cryptography Architecture* (JCA) and the *Java Cryptography Extensions* (JCE) support cryptography in Java. Java's cryptography support is spread over the JCA and JCE libraries due to the USA government's cryptography export control laws that were in effect at the time Java was first released. The JCA contains interfaces and algorithms that can be exported without restrictions while the JCE implementations are subject to export controls.

The JCA and JCE are designed using a modular provider-based architecture with API interfaces and classes defined in `java.security`, `javax.crypto` packages are included in the standard Java distribution. Implementations of the JCA and JCE APIs are packaged as providers and added to the Java Runtime Environment (JRE).

Developers write application code to interact with the standard API layer which then routes the calls to the algorithm implementations in the installed providers (figure 4.6).

Figure 4.6 Application code calls the standard JCA and JCE APIs making the application code independent of the implementation of the crypto algorithm. The JCA and JCE layer routes call to a provider's implementation. Java 11 based OpenJDK ships with 13 providers out of the box.



How USA export control laws affected the design of the Java crypto libraries

The SunJCE^[4] provider is the default provider shipped with the majority of Java distributions derived from OpenJDK. In Java 8 and earlier, the default SunJCE provider was configured with weak algorithms to comply with export controls. An unlimited strength JCE extension .jar was available for download as an add-on to the Java Runtime Environment to enable strong cryptography for Java users in countries friendly to the American government.

The Bouncy Castle[\[5\]](#) JCE provider was created by Australian developers who are not subject to USA export control laws and can offer strong cryptography support to all Java users. Bouncy Castle was also optimized for embedded devices with low power and CPU. Bouncy Castle implements more algorithms than what ships with the default SunJCE provider.

Cryptography export control laws in the USA were relaxed, Java 9 (released in 2017) and later ships with unlimited strength encryption.

A *Hardware Security Module* (HSM) implements cryptography algorithms in hardware to increase security. HSMs are used in highly sensitive secure environments such as financial and government applications. Many HSM vendors ship JCE implementations to make their HSM modules usable from Java code without developers having to learn the vendor's proprietary API. Amazon offers a cloud-based HSM, they ship CloudHSM[\[6\]](#) JCE provider for users to add to Java applications deployed to AWS.

Multiple providers can be installed in the same JRE, which means that there can be multiple implementations of the same algorithm for a developer to choose from. The Java 21 OpenJDK bundles 13 security providers by default. Developers access algorithms using names that are standardized and documented[\[7\]](#) in the JCA. For example, "SHA-256" is the JCA standard name for the SHA-2 hashing algorithm with a 256-bit output.

Cryptographic hash functions are sometimes referred to as message digests because they take an arbitrary-sized input and turn it into a fixed-size output. The Java Cryptography Architecture (JCA) uses the `java.security.MessageDigest` abstract class to define the API for working with cryptographic hash functions.

You can obtain an instance of `MessageDigest` using the static method `MessageDigest.getInstance()`. For example, `MessageDigest.getInstance("SHA-256")` is used to request an implementation of the SHA-256 hash function from the default provider installed in the JRE.

Listing 4.1 shows a simple implementation of SHA with plain Java. As you can observe the code is straightforward since the language provides the

needed tools to apply the basic cryptographic functions. You find this example in project ssfd_ch4_ex1 provided with this book.

Listing 4.1 Using SHA-3 to get a hash for a given file

```
static String sha3HexOfResource(String resourceName)
    throws IOException, NoSuchAlgorithmException {
    try (InputStream is = #A
        Main.class.getClassLoader().getResourceAsStream(resourceName)) {
        if (is == null) {
            throw new IllegalArgumentException(
                "Error: Resource '" + resourceName +
                "' not found on classpath.");
        }
        byte[] data = is.readAllBytes(); #B
        byte[] digest = MessageDigest.getInstance("SHA3-256")
            .digest(data); #C
    }
    return HexFormat.of().formatHex(digest); #D
}
```

Running the project, you'll get the following output as presented in the next snippet. The string starting with 542b49... is the SHA-3 encoding of the refunds.json file that you can find in the resources folder of the project. You can try to apply the same code to different files and compare the differences in the result. Run it multiple times for the same file. You will observe the result is always the same for the same file.

```
SHA3-256(refunds.json) =
542b49a04073502046b0de5751d5f7a9afe7c1ff2d63524c0863233a53de0c14
```

The `java.security.MessageDigest` class provides access to a variety of industry-standard hashing algorithms. The MD2, MD5, and SHA-1 are insecure hashing algorithms supported by Java for backward computability reasons, they should only be used to work with historical data. SHA-2 and SHA-3 algorithms are considered secure and should be used for any new application development.

The “Java Security Standard Algorithm Names” contains a list of all the algorithm names that can be passed to `MessageDigest.getInstance()` method. SHA-224, SHA-256, SHA-384, SHA-512 refer to the SHA-2 algorithm with different output sizes. The SHA-3 algorithm of various output sizes can be looked up using the name SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

TIP

The JCA and JCE are powerful generic low-level APIs but are not developer-friendly. It is easy to misconfigure or misuse them accidentally. The rest of this chapter and the subsequent use JCA and JCE because they are the standard libraries in Java and are always available to every Java program. You should be able to read code written in JCA and JCE. However, for production code, you should use a developer-friendly crypto library such as Google Tink[\[8\]](#).

Google Tink is designed to reduce common programming errors when working with cryptography libraries. Tink provides Java, C++, Objective-C, Go, Python, and JavaScript implementations, which means you can learn the library once and use it in multiple programming languages. The Java version of Tink is built as a developer-friendly layer on top of JCA and JCE with some extra features that are not available in the core Java libraries.

Listing 4.2 shows how the same encoding can be done using Google Tink library. You find this example in project ssfa_ch4_ex2.

Listing 4.2 Hashing the refunds.json file using Google Tink

```
public static String hashRefundsJsonWithSha3(String resourceName)
    throws IOException, GeneralSecurityException {

    try (InputStream in = #A
        Main.class.getClassLoader()
        .getResourceAsStream(resourceName)) {

        MessageDigest md = #B
        EngineFactory.MESSAGE_DIGEST.getInstance("SHA3-256");

        byte [] fileContent = in.readAllBytes(); #C
```

```

        byte [] digested = md.digest(fileContent); #D
        return HexFormat.of().formatHex(digested); #E
    }
}

```

Running the project you will observe it produces the same result as the previous application we worked on in this chapter – ssfd_ch4_ex1.

```
SHA3-256(refunds.json) =
542b49a04073502046b0de5751d5f7a9afe7c1ff2d63524c0863233a53de0c14
```

Cryptographic algorithms are CPU intensive. As you scale your applications to handle more users you will require more CPU cores, which increases operational costs. The OpenJDK provides fast, portable implementations of cryptographic algorithms in Java. However, low-level optimization techniques are possible to do in C and assembly language but not in Java. These optimizations can speed up the performance of cryptography operations at the expense of more complex deployment.

Amazon and Google offer two highly optimized JCA providers:

- Amazon Corretto Crypto Provider (ACCP)
- Google Conscrypt[\[9\]](#)

Amazon Corretto Crypto Provider (ACCP) is a JCA implementation built on top of the highly optimized OpenSSL libcrypto native C library. Amazon claims that “AWS Snowball uses ACCP to run cryptographic functions about 20 times faster, doubling its data transfer speed. Amazon S3 and AWS IoT use ACCP to enable new cryptographic features previously too resource-intensive to deliver.” [\[10\]](#) .

Google Conscrypt is a JCA provider implemented on top of BoringSSL[\[11\]](#) which is Google’s fork of the OpenSSL native library. One of the main advantages of Conscrypt is that it supports Android devices and OpenJDK.

It is easy to add the Amazon Corretto Crypto Provider, or the Google Conscrypt provider into your deployment without having to rewrite your code. Therefore, starting with the default JCA provider implementations that ship with OpenJDK is best. You can switch providers once the cost savings

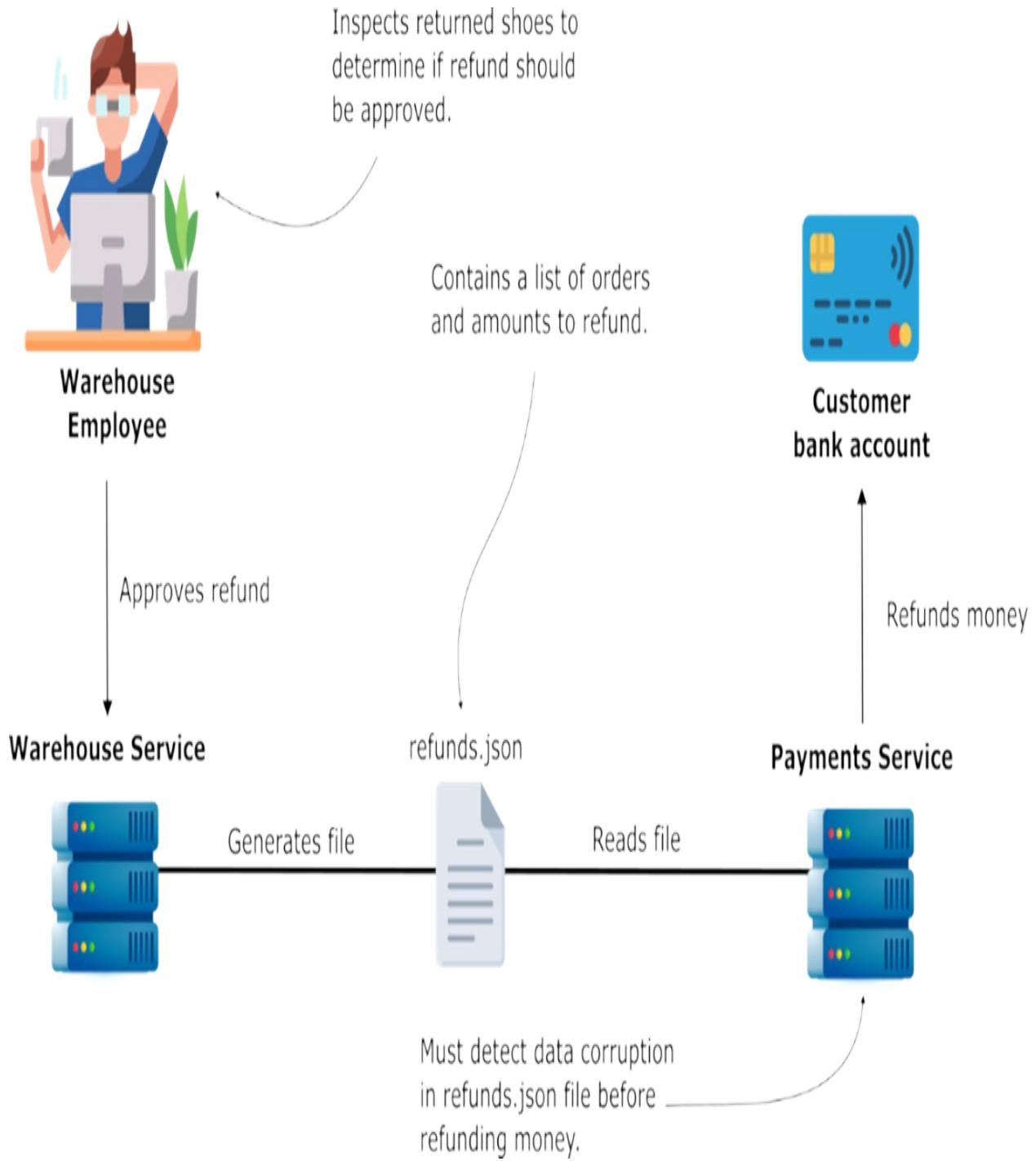
from a non-default provider outweighs the complexity of adding it your deployment pipeline.

4.4 Implementing message integrity in Java

It has taken us a few pages to learn about the concept of cryptographic hash functions and the Java cryptography architecture. We are now ready to implement the ACME Inc. scenario discussed earlier. The scenario is reproduced below, so you don't have to flip to earlier pages.

ACME Inc. customers mail newly purchased shoes they don't like to the ACME warehouse for a refund. The warehouse staff use the warehouse management application to authorize refunds once they verify that the returned shoes are in good condition. Once per day, the generated refund file is sent to the payment service for processing, as shown in figure 4.7.

Figure 4.7 ACME Inc. staff approve refunds using the warehouse management application. Once a day, the warehouse management application generates a refunds.json file. The payment service issues refunds to customer credit cards for the amount specified in the refunds.json file.



The sample application discussed in this section implements the ACME Inc. data corruption detection scenario. You can find the code at <[link](#)>. The GitHub repo contains detailed instructions explaining how to run the application on a developer's laptop. I highly recommend that you run the sample application to deepen your understanding of the topic. There are two projects in the sample repo ssfd_ch4_ex3-warehouse and ssfd_ch4_ex3-

payments:

- *Warehouse*, a Java application that writes out the `refunds.json` file and `refunds.json.sha256` containing the hash value of the `refunds.json` contents.
- *Payments*, contains a Spring Boot application that reads the `refunds.json` file, computes the SHA-256 value of its contents, then compares the computed value against the expected value stored in the `refunds.json.sha256`. If the values match the `refunds` file is processed (values are displayed in the endpoint response) otherwise an exception is thrown.

Start by running the *Warehouse* application which is a plain Java app. The project already provides a demo `refunds.json` file in the resources folder. Running the app will generate another file containing the SHA3 hash of the `refunds.json` file. The way it does the hashing is the one we discussed previously.

The payments service use the `Refund` record in the next snippet as the Java representation of refundable orders.

```
public record Refund(String orderId, BigDecimal amount) {}
```

I kept the `Refund` record to a bare minimum so we could demonstrate the cryptographic hash function concepts without making the sample application complex. The *Payments* app implements an endpoint which receives both the `refunds.json` file and its hash. You find the endpoint's implementation in listing 4.3.

Listing 4.3 The refunds endpoint

```
@RestController
@RequestMapping("/api/refunds")
@AllArgsConstructor
public class RefundController {

    private final RefundService refundService;
```

```

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                  produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Refund> uploadRefunds(
        @RequestHeader("X-Content-SHA3") String sha3, #A
        @RequestBody byte[] body) {
        return refundService.verifyAndReturnRefunds(body, sha3); #B
    }
}

```

Listing 4.4 shows a simple implementation of the refunds service. The service method takes a file input along with its hash. It then recalculates the hash of the received data and compares the result with the provided hash value. If the two match, the data is considered valid and the application returns the orders in the response. If the hash is missing, or if the computed and provided hash values do not match, the application throws an exception.

Listing 4.4 The service which validates the received file by computing its hash

```

@Service
@AllArgsConstructor
public class RefundService {

    private final ObjectMapper objectMapper;
    private final HashManager hashManager;

    public List<Refund> verifyAndReturnRefunds(
        byte[] bodyBytes, String providedHashHex) {

        if (providedHashHex == null || providedHashHex.isBlank()) {
            throw new InvalidHashException(
                "Missing X-Content-SHA3 header");
        }

        String computed = hashManager.computeSha3Hex(bodyBytes);
        if (!computed.equalsIgnoreCase(providedHashHex.trim())) {
            throw new InvalidHashException(
                "#A\n" + "Invalid SHA3 hash for provided refunds file")
        }

        try {
            return objectMapper.readValue(
                bodyBytes, new TypeReference<List<Refund>>(){}));
        } catch (IOException e) {
            throw new IllegalArgumentException(
                "Invalid refunds JSON format", e);
        }
    }
}

```

```
    }  
}  
}
```

Now it's time to try the example yourself! Run the Payments project and call the endpoint. You can find detailed instructions on how to do this in the project's README file. First, provide the file together with the hash value generated by the Warehouse application. In this case, calling the endpoint will succeed without any exceptions. Next, modify the contents of the refunds.json file but do not recalculate the hash. When you send the request again, the endpoint will throw an exception.

By computing the SHA-256 hash of the refunds.json file and comparing it with the expected hash stored in the refunds.json.sha256 file, the payment service can detect accidental data corruption. However, we can't detect if a hacker has changed the refunds.json and generated a corresponding refunds.json.sha26. Detecting active tampering is the focus of the next section.

4.5 Message Authentication Code (MAC)

In the ACME Inc. refunds scenario implemented in the previous section, we detected accidental data corruption. However, a hacker can fool the payment service into refunding the wrong amount by modifying the refunds.json and the associated refunds.json.sha256 files. To stop this type of attack, the payment service must validate two things before processing the refunds:

- *Authenticity*, the refunds.json file was created by the warehouse service and not a hacker.
- *Integrity*, the refunds.json file was not modified after it was created.

We can extend the integrity detection algorithm from the last section to construct a *Message Authentication Code* (MAC) that can guarantee integrity and authenticity. The MAC can only be computed by the sender and receiver of a message. A hacker can view and modify the message body, but they cannot compute the MAC. A MAC enables the following data exchange:

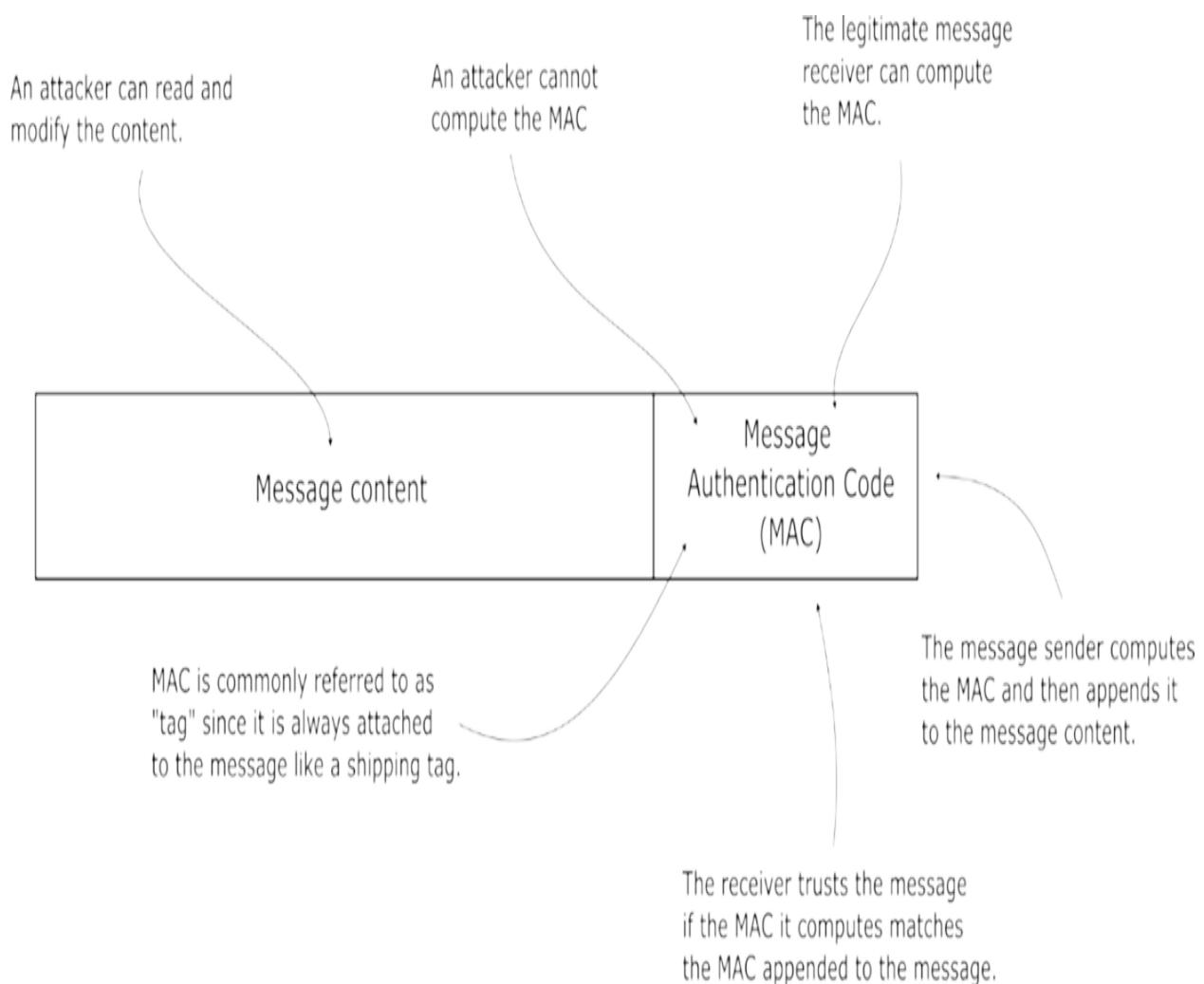
- Sender, generates the message body, computes the MAC on the message

body, sends the message and MAC to the recipient.

- Recipient, receives the message body, computes the MAC on the received message body, compares the computed MAC against the expected MAC, if the computed and expected MAC match the message came from the sender and is unmodified.

Figure 4.8 visually describes the MAC.

Figure 4.8 Message Authentication Code (MAC) is data added to the end of a message that can be used by the receiver of a message to identify who sent the message and that the message was not tampered with.



The sender, the receiver, and a hacker can compute the cryptographic hash of a message's content because all you need is the message body to compute the hash value. However, computing the MAC requires the message body and a

secret key. The secret key is known to the sender and receiver but not the hacker, so the hacker cannot compute a MAC. A MAC can be built out of a cryptographic hash function as shown in figure 4.9.

Figure 4.9 Computing the Message Authentication Code (MAC) requires both the secret key and the input message. When the hacker modifies the input, they cannot compute the MAC because they don't know the secret key that is part of the computation. Since they can't compute the MAC they can't fool the receiver with a fake message body.

Only the owner of the input and the party who needs to verify the MAC have the secret key.



A hacker may have access to the input.

Input message

Secret key

The key is used during the cryptographic operation. Small changes of the input would produce completely different output results.

Cryptographic Hash Function



Hacker can't perform the cryptographic operation since they don't have the secret key.

**message authentication code
(MAC)**

There are many algorithms for implementing a MAC. Building the MAC using a cryptographic hash function as illustrated in figure 4.9, is a popular approach. However, you must be careful how the secret key is mixed with the input.

A simple approach for mixing the secret key into the input, is to concatenate the message body with the secret key then compute the hash of the resulting string. For example, if the input is “abc”, the secret key is “123”, and the hash function is SHA3-512, then the MAC value is the result returned by SHA3-512(“abc123”).

WARNING

The concatenation approach for mixing the secret with input is secure with some hash functions and insecure with others. For example, using SHA-2 with the concatenation approach is unsafe, while using SHA-3 is secure. HMAC, described in the following section, is a secure way to build a MAC from a hash function.

4.5.1 Hashed Message Authentication Code (HMAC)

Hashed Message Authentication Code (HMAC) is a widely used approach for implementing message authentication code using a cryptographic hash function and a secret key. HMAC works with SHA-2, SHA-3 and other hash functions. The output of the HMAC function is exactly the same size as the output of the hash function it is configured to use. For example, an HMAC based on SHA-256 will produce 256-bits of output.

The details of how the HMAC algorithm mixes the secret key with the input is beyond the scope of this book. Luckily, programming language libraries have excellent out-of-the-box HMAC implementations, so you don’t need to know the details of how the HMAC algorithm works. You just need to know when and how to use it.

TIP

In cryptography literature \parallel is used to indicate concatenation. For example, “abc” \parallel “123” results in the string “abc123”. You will see the \parallel concatenation notation used frequently in this book. A common mistake is to assume that an HMAC is computed by first concatenating the input bytes with the secret key and then hashing the result. For example, if the input is “abc” and the secret key is “123” then SHA256(“abc123”) is not equal to HMAC(“abc”, “123”, SHA256). HMAC uses a more elaborate algorithm.

4.5.2 Java Support for HMAC

Java 21 ships with support for computing HMAC using the MD5, SHA-1, SHA-2, and SHA-3 hash functions. Since MD5 and SHA-1 are insecure, they should only be used when processing historical data that uses these old hash functions.

The code in listing 4.8 shows a simple utility function that takes a byte array and key to compute the HMAC of the input bytes using SHA-256, and the key returns the result as a hexadecimal string. You find this in project ssfd_ch4_ex4 provided with the book.

Listing 4.5 Computing an HMAC using SHA-256 in Java

```
public static byte[] hmacSha256(byte[] key, byte[] data) throws E
    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(new SecretKeySpec(key, "HmacSHA256"));
    return mac.doFinal(data);
}
```

The `javax.crypto.Mac` class provides access to variety of industry standard HMAC algorithms. “HmacSHA256” is the standard name for an HMAC based on SHA-256. The “Java Security Standard Algorithm Names”[\[12\]](#) contains a list of all the algorithm names that can be passed to `Mac.getInstance()` method. The `SecretKeySpec` class is a versatile way to store a key's bytes along with the algorithm it is designed for. Its generic design allows it to be used with a variety of algorithms.

The key for the HMAC function should contain enough randomness such that it is not easy to guess. A 256-bit key for SHA-256 is a reasonable default to

go with. Since the HMAC key is effectively a password you should treat it with care to make sure it is not stolen. Your corporate information security team might have standards and recommendations on the minimum size of an HMAC key and which hash function to use with an HMAC. You should consult them before using an HMAC in your application.

HMAC is a key building block in many commonly used industry standard protocols such as Transport Layer Security (TLS) and JSON Web Token (JWT). Developing an intuitive understanding of what HMAC is and how it is used will make understanding security documentation significantly easier.

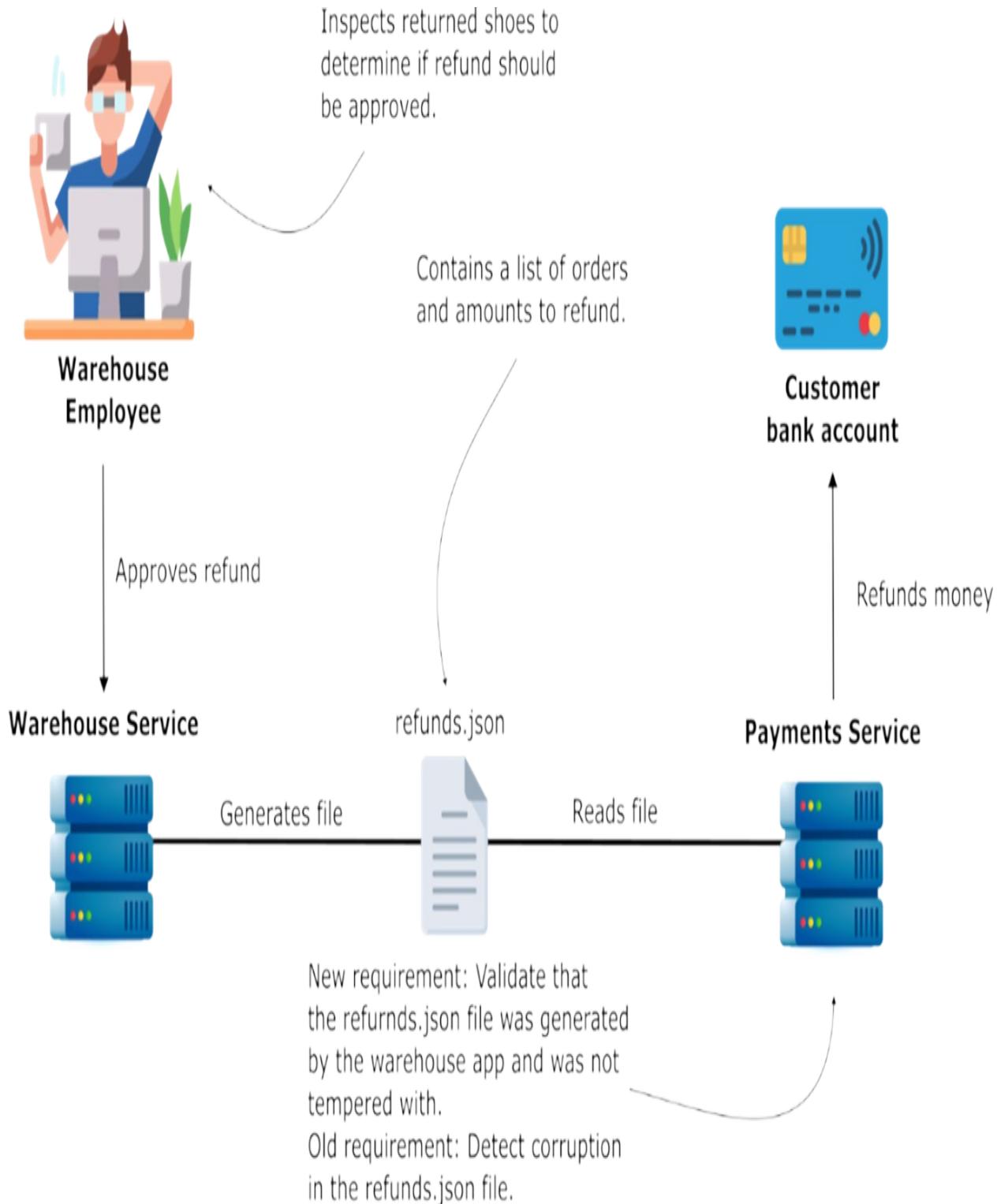
4.5.3 Exercises

4. What extra guarantee does HMAC provide over a plain hash function?
5. Why is concatenating a secret key with a message and hashing it (e.g., SHA256("message" + "secret")) not the same as HMAC?
6. Which cryptographic goals do SHA-2, SHA-3, and HMAC each fulfill?
7. What risk exists if you hardcode HMAC secret keys in configuration files?
8. In the ACME refunds scenario, what happens if the refunds.json file is changed but the HMAC doesn't match?

4.6 Guaranteeing authenticity using HMAC

In this section, we will review the implementation of the ACME Inc. scenario using an HMAC in Java. Recall that the warehouse application sends a refunds.json file to the payments service so that payment can issue refunds to customer credit cards. The payments service wants to ensure that the warehouse application created the refunds.json and that it was not tampered with after it was created (figure 4.10).

Figure 4.10 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service issues refunds to customer credit cards for the amount specified in the refunds.json file.



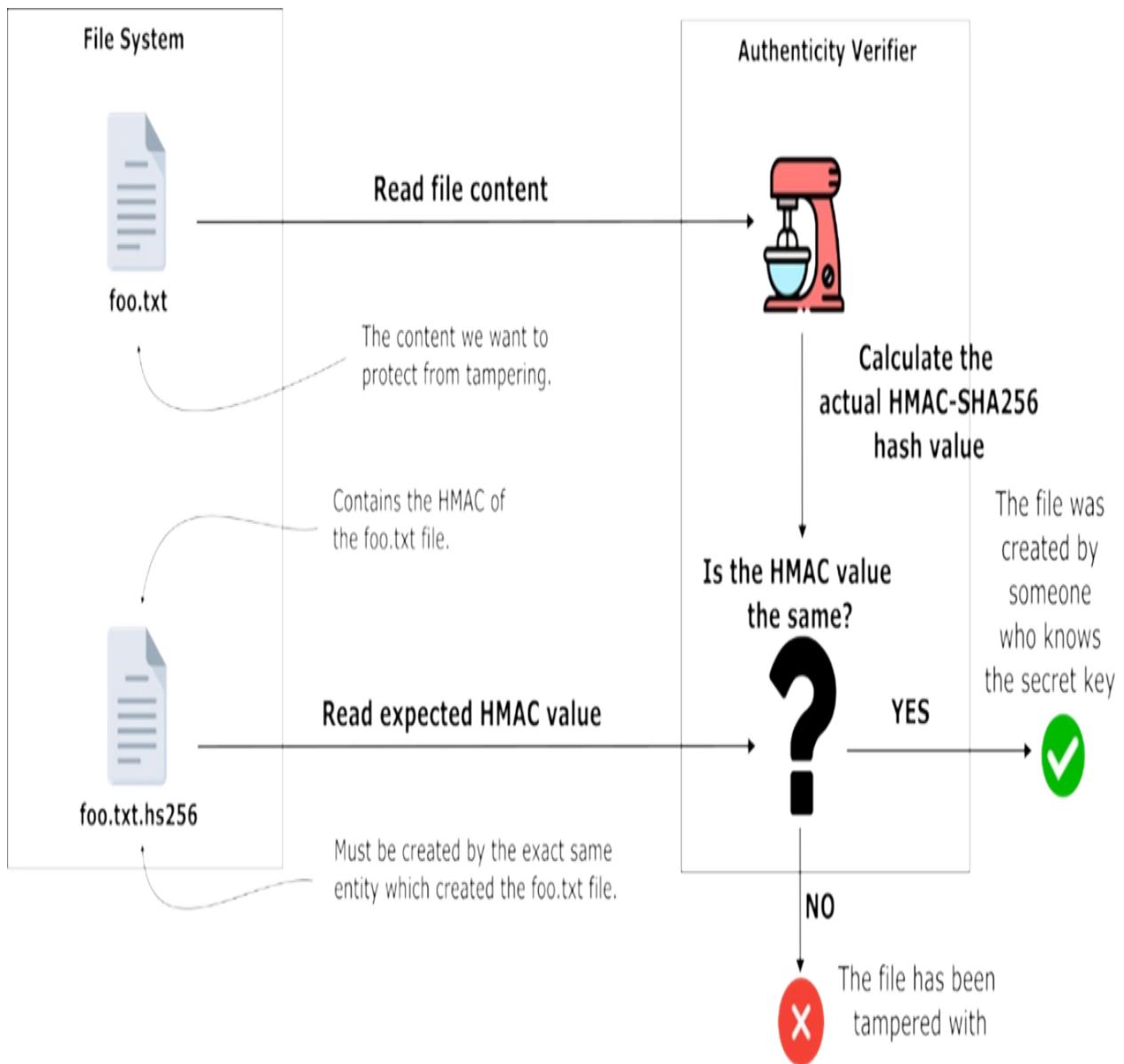
When the payment service retrieves the `refunds.json` file, it computes the HMAC-SHA256(`refunds.json`, secret key) and then compares it to the expected value in `refunds.json.hs256`. If the values match, then there was

no data corruption and the creator of the `refunds.json` and `refunds.json.hs256` must have known the secret key required to compute the HMAC. Since the secret key is only known to the warehouse application and the payments service, the payment service can assume that the `refunds.json` file was created by the warehouse application.

You can find this example implemented in the projects `ssfd_ch4_ex5-warehouse` and `ssfd_ch4_ex5-payments` provided with this book. The Warehouse project is a plain Java application that calculates the HMAC for a `refunds.json` file stored in its resources folder. This application simulates the request payload generated by the Warehouse service in the system. The Payments project is a Spring Boot application that exposes an endpoint. It receives the refunds file in the HTTP request body, validates the HMAC, and processes the refunds — in our simplified model, this just means returning them in the HTTP response body.

Figure 4.11 visually explains the validation flow .

Figure 4.11 Determine who created a file and that the file was not tampered with using a Hashed Message Authentication Code (HMAC). The producer of the `foo.txt` file computes the HMAC-SHA-256 value of the content and writes it to `foo.txt.hs256`. A consumer computes the SHA-256 value of the contents of `foo.txt` then compares the computed value to the expected value in `foo.txt.hs256`. If the computed and expected HMAC values differ, then `foo.txt` has been tampered with and an error should be raised.



An HMAC requires a *secret key* to be shared between the warehouse application and the payments service. How do the warehouse application and the payment service agree on the value of this secret key? For now, we simplify the assumption that the owner of the payments and warehouse service met in person and came up with a secret key during the meeting. Meeting in person to agree on a shared key is problematic, we will learn how to solve the key agreement problem in a later chapter.

WARNING

Storing a key in the configuration file in clear text (and especially hardcoded) is a security anti-pattern. The book is dedicated to securely storing secrets in credential storage services. But to simplify the examples and allow you focus on the subject to learn, in the book examples we'll always store the secrets together with the example.

Similar to what we have done in section 4.4, first you need to run the Warehouse application to generate the `refunds.json.hs256` file. Then, you start the Payments project and call the `/api/refunds` endpoint (details about how to do this in the project README file).

By computing the HMAC-SHA256 hash of the `refunds` file and comparing it with the expected hash stored in the `refunds.json.hs256` file payment service can detect data corruption and ensure that the file was generated by someone with the secret key used to generate the message authentication code. Listing 4.6 shows the service method implementation of our example project.

Listing 4.6 The service validates the HMAC before processing the refunds

```
public List<Refund> verifyAndReturnRefunds(
    byte[] bodyBytes, String providedHmacHex) {

    if (providedHmacHex == null || providedHmacHex.isBlank()) {
        throw new InvalidHashException("Missing X-Content-HMAC header");
    }

    String computed = hmacManager
        .computeHmacHex(bodyBytes); #A
    if (!computed.equalsIgnoreCase(providedHmacHex
        .trim())) { #B
        throw new InvalidHashException(
            "Invalid HMAC for provided refunds file");
    }

    try {
        return objectMapper.readValue(
            bodyBytes, new TypeReference<List<Refund>>(){}));
    } catch (IOException e) {
        throw new IllegalArgumentException("Invalid refunds JSON form");
    }
}
```

HMAC is widely used in many security protocols, including Transport Layer Security (TLS). The knowledge you've gained in this chapter will help you better understand various security protocols and libraries. Table 4.5 highlights the cryptographic goals that HMAC fulfills.

Table 4.5 Cryptography goals and algorithms matrix

Goal	SHA-2	SHA-3	HMAC
Integrity	Yes	Yes	Yes
Authentication	No	No	Yes
Confidentiality	No	No	No
Non-repudiation	No	No	No

Give yourself a pat on the back. You have learned a lot about cryptographic hash functions and message authentication codes.

TIP

Even though HMAC does not provide content confidentiality it is still a critical component in the security toolbox. You will see some applications of HMAC in the upcoming chapters.

4.7 Exercise answers

1. What are the four main goals of cryptography?
Integrity (data isn't changed), Authentication (we know who sent it), Confidentiality (only the right people can read it), and Non-repudiation (senders can't deny sending something).
2. How is a cryptographic hash function different from the hashCode() method in Java?
A cryptographic hash function produces a long, fixed-size output (like 256 bits) designed for security, with properties like one-way and collision resistance. The Java hashCode() method just returns a 32-bit integer for use in hash tables and is not secure for cryptography.
3. What problem does using SHA-256 on a file solve, and what problem does it not solve?
It detects accidental corruption (like network errors or disk corruption)

because any change in the file changes the hash. But it does not prevent active tampering — an attacker could change both the file and the hash.

4. What extra guarantee does HMAC provide over a plain hash function?
HMAC ensures authenticity in addition to integrity. Only parties with the secret key can generate or validate the HMAC, so attackers can't forge it.
5. Why is concatenating a secret key with a message and hashing it (e.g., SHA256("message" + "secret")) not the same as HMAC?
Because HMAC uses a carefully designed algorithm to mix the key and message. Simple concatenation is insecure with some hash functions (like SHA-2), while HMAC ensures security across algorithms like SHA-2 and SHA-3.
6. Which cryptographic goals do SHA-2, SHA-3, and HMAC each fulfill?
 - SHA-2 and SHA-3: Integrity only.
 - HMAC: Integrity + Authentication.
 - None of them provide Confidentiality or Non-repudiation.
7. What risk exists if you hardcode HMAC secret keys in configuration files?
Attackers who gain access to the code or config can steal the secret key.
Best practice is to store keys in a secure credential storage system, not in plain text.
8. In the ACME refunds scenario, what happens if the refunds.json file is changed but the HMAC doesn't match?
The payments service will reject the file, throwing an exception, and no refunds will be processed.

4.8 Summary

- Cryptography solves four fundamental security problems: integrity (data unchanged), authentication (sender identity), confidentiality (data privacy), and non-repudiation (legal proof of transactions).
- Cryptographic hash functions like SHA-2 and SHA-3 produce fixed-size outputs from any input and have one-way properties and collision resistance, making them ideal for detecting data corruption.
- Hash functions are deterministic, always producing the same output for the same input, but different inputs produce completely different outputs.

- SHA-1 and MD5 are deprecated due to security vulnerabilities, while SHA-2 and SHA-3 are currently considered secure for new applications.
- Cryptographic hash functions can detect accidental data corruption but cannot prevent active tampering by attackers who modify both data and hash files.
- Message Authentication Code (MAC) requires a secret key shared between sender and receiver to guarantee both integrity and authenticity of messages.
- Hashed Message Authentication Code (HMAC) is a secure way to build MAC using cryptographic hash functions and secret keys, widely used in protocols like TLS and JWT.
- Java Cryptography Architecture (JCA) and Java Cryptography Extensions (JCE) provide cryptographic support through a provider-based architecture with standardized APIs.
- Simple concatenation of secret keys with messages before hashing is insecure with some algorithms, while HMAC provides a secure mixing approach for all supported hash functions.
- Applications should design for hash function upgrades since algorithms become insecure over time, requiring version fields and migration strategies.
- HMAC enables detection of both accidental corruption and active tampering, as attackers cannot forge valid HMAC values without knowing the secret key.
- Secret keys for HMAC should be stored securely in credential management systems rather than hardcoded in configuration files to prevent theft.

[1] <https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html>

[2] <https://issues.sonatype.org/browse/MVNCENTRAL-5276>

[3] <https://lwn.net/Articles/811068/>

[4] <https://www.oracle.com/java/technologies/javase-jce8-downloads.html>

[5] <https://www.bouncycastle.org/>

[6] <https://docs.aws.amazon.com/cloudhsm/latest/userguide/java-library-install.html>

[7] <https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html>

[8] <https://github.com/google/tink>

[9] <https://conscrypt.org/>

[10] <https://aws.amazon.com/blogsopensource/introducing-amazon-corretto-crypto-provider-accp/>

[11] <https://boringssl.googlesource.com/boringssl/>

[12] <https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html>

5 Advanced Encryption Standard

This chapter covers

- Using Advanced Encryption Standard (AES) to protect data confidentiality
- Selecting a safe AES operating mode for typical application development needs
- Using AES in Galois Counter Mode (GCM) to provide data integrity, authenticity, and confidentiality

Users expect applications to protect their data and keep it confidential according to the laws where they live. For example, European Union (EU) citizens expect applications to comply with the *General Data Protection Regulation* (GDPR) law. Encryption is needed in most applications because most countries have laws governing data confidentiality, as a developer you must be able to use encryption to protect user data.

The Advanced Encryption Standard (AES) is the most widely used technology for ensuring data confidentiality. All the public cloud providers including Amazon, Google and Microsoft use the Advanced Encryption Standard (AES) extensively to secure their APIs and services. Windows, Linux, and MacOS use AES for disk encryption. Foundational networking protocols such as *Internet Protocol Security* (IPsec), *Transport Layer Security* (TLS), *Secure Shell* (SSH) protocol, all leverage AES to deliver security.

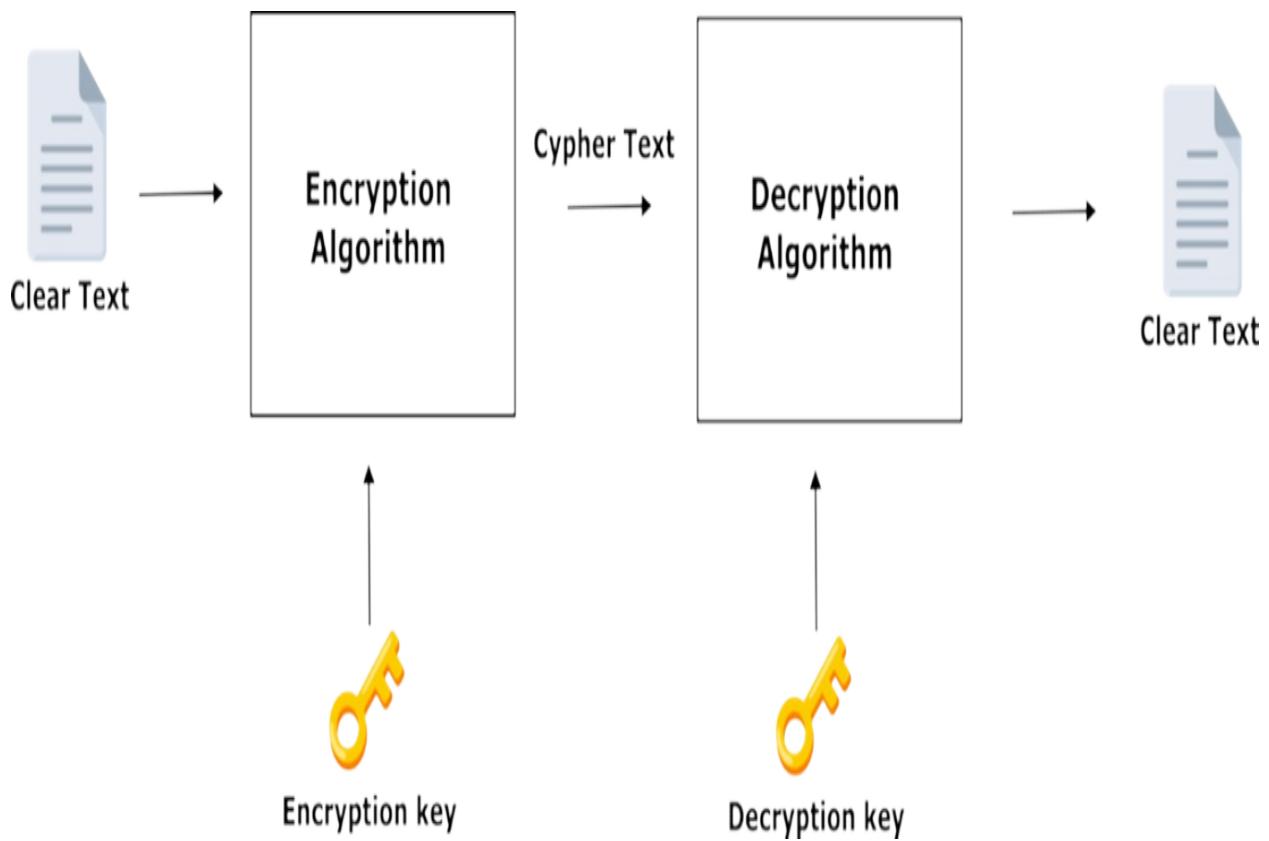
Working with AES is a critical security skill for an application developer. This chapter provides a developer friendly introduction to AES. We will not cover the mathematical details that underpin AES. Instead, we will cover how to use AES through a series of Java sample applications to provide you with the intuition to use AES successfully.

5.1 Advanced Encryption Standard Overview

To keep data safe during transmission or storage, it needs to be encrypted. Encryption scrambles the data using a key, making it look like a random sequence of bits. This ensures that no one can understand the data without the proper key.

Decryption is the process of reversing the scrambling to turn the encrypted data back into its original form. The original data is called plain text, while the scrambled, encrypted version is called cipher text. Figure 5.1 depicts the relationship between encryption and decryption.

Figure 5.1 A cipher consists of an algorithm for encrypting and decrypting data using a key. The details of encryption and decryption algorithm are public knowledge while the key is kept secret. If the same key is used to encrypt and decrypt the cipher is called a symmetric cipher otherwise it is called an asymmetric cipher.



Definition

Encryption is the process of transforming readable data (plaintext) into an unreadable form (ciphertext) using a key, so that only someone with the right

key can turn it back into its original form.

Definition

Decryption is the process of transforming encrypted data (ciphertext) back into its original readable form (plaintext) using the correct key.

Encryption algorithms can be classified into two families: *symmetric* and *asymmetric*.

Symmetric ciphers use the same key to encrypt and decrypt data they will be covered in this chapter. Asymmetric ciphers use a pair of keys one for encrypting data and another for decrypting data, they will be covered in chapter 6.

Designing a secure encryption algorithm for use in industry and government on a wide range of devices, from low-power IoT devices to phones and supercomputers, is a huge engineering effort. The Advanced Encryption Standard (AES) is a very popular encryption algorithm.

It is the de facto standard for symmetric key encryption algorithms used in all applications, including cloud-native applications. All major CPU architectures—Intel, ARM, and others—provide hardware support and acceleration for AES, so you don't need to worry about performance when using AES.

History of AES

In 1997 the U.S. National Institute of Standards and Technology (NIST) ran an open competition to select an “an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century”. The competition attracted 15 submissions which were put through a rigorous analysis to select a winner. Rijndael an algorithm designed by two Belgian academics Vincent Rijmen and Joan Daemen in 1998 was selected as the AES standard in 2001. Before AES the *Data Encryption Standard* (DES) was the official encryption algorithm of the USA government it was designed by IBM and the USA cyber spies the *National Security Agency* (NSA). A lot of researchers and originations outside the

USA were suspicious that the NSA had engineered a weakness into DES. The open process used to select the winning AES algorithm contributed to confidence in the security of AES even by those who are suspicious of the NSA's intentions.

The AES algorithm requires its input to be precisely 128-bit or 16-bytes long. If the input is shorter than 128 bits, extra input bits must be added to make the input exactly 128-bits. Adding extra input bits is called *padding*. If the input is longer than 128 bits it must be broken up into series of 128-bit blocks.

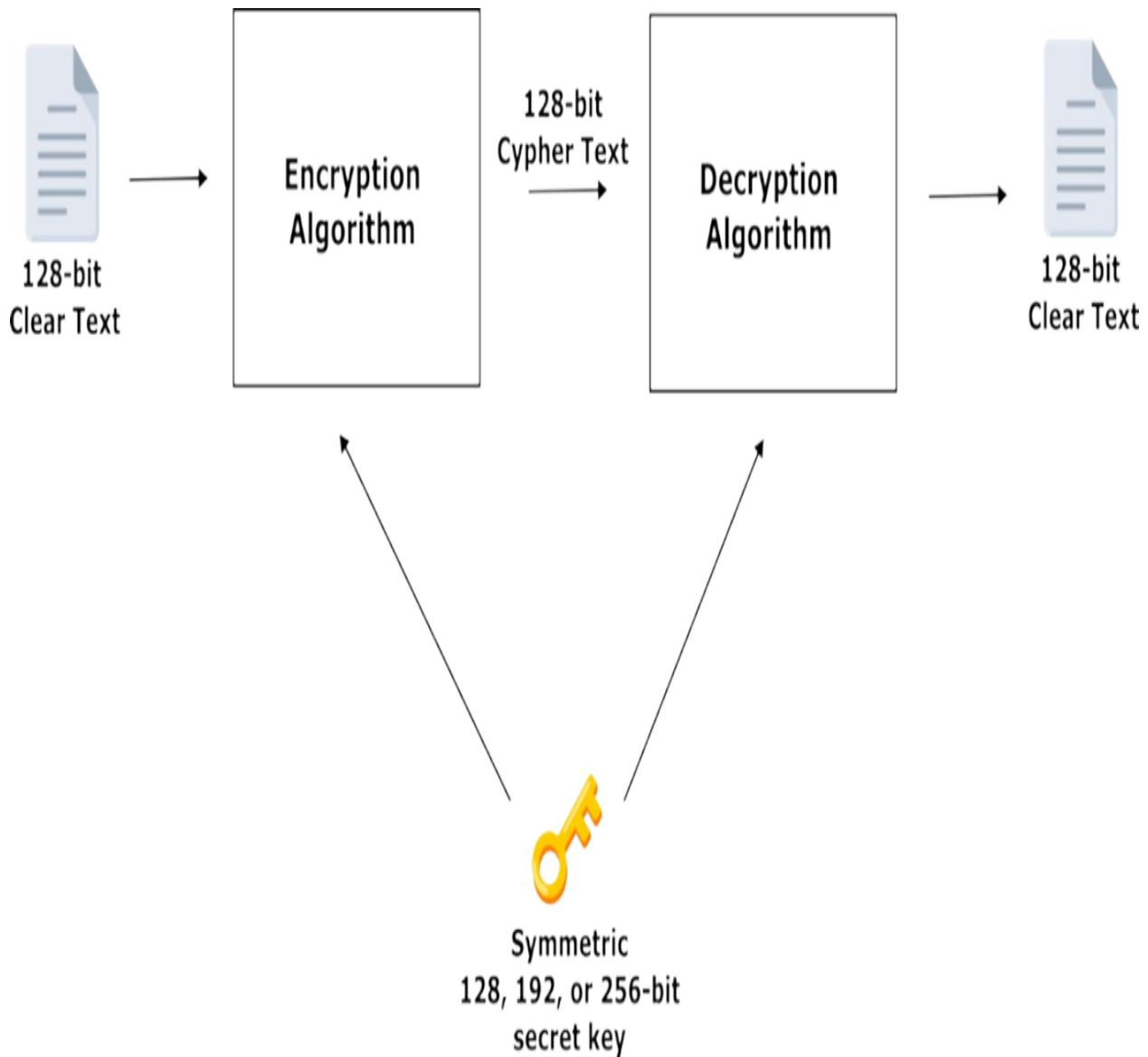
Encryption algorithms like AES, which work on a fixed-size input, are called *block ciphers*.

Definition

A block cipher algorithm is a method of encrypting data by dividing it into fixed-size blocks (like 128 bits) and transforming each block into ciphertext using a secret key.

Since AES uses the same key to encrypt and decrypt data it is classified as a symmetric block cipher (figure 5.2). AES encryption keys can be 128, 192, 256 bits.

Figure 5.2 AES is a block cipher because it requires its input be a fixed size 128-bits long. AES is a symmetric cipher because the key used to encrypt is the key used to decrypt.



Warning

The AES may use multiple configuration options to choose from. Unfortunately, some AES configurations are insecure and should not be used. It is critical to use a secure configuration of AES. How to determine if a particular AES configuration is secure? The rest of this chapter provides you with an overview of the key configuration settings of AES, along with recommendations that are considered safe today. Corporate security teams publish recommendations for AES configurations developers should use when working with AES. Consult with your information security team for recommended AES configurations.

5.2 Modes of operation

The mode of operation is the most complex and essential topic for an application developer to understand AES. I have broken up this section into small subsections to make the learning journey easier. You might want to come back and review this section once you have finished reading the chapter.

Suppose we want to encrypt a 1605-byte message with AES. Since AES only works on 16-byte blocks, we must break up the message into 1001 blocks. 1000 16-byte blocks plus a final block with 5 bytes of message content and 11 bytes of padding. We then need a way to apply AES to each of the 1001 blocks.

The algorithm for breaking up input into blocks of 16 bytes and applying AES to each block is called the *mode of operation*.

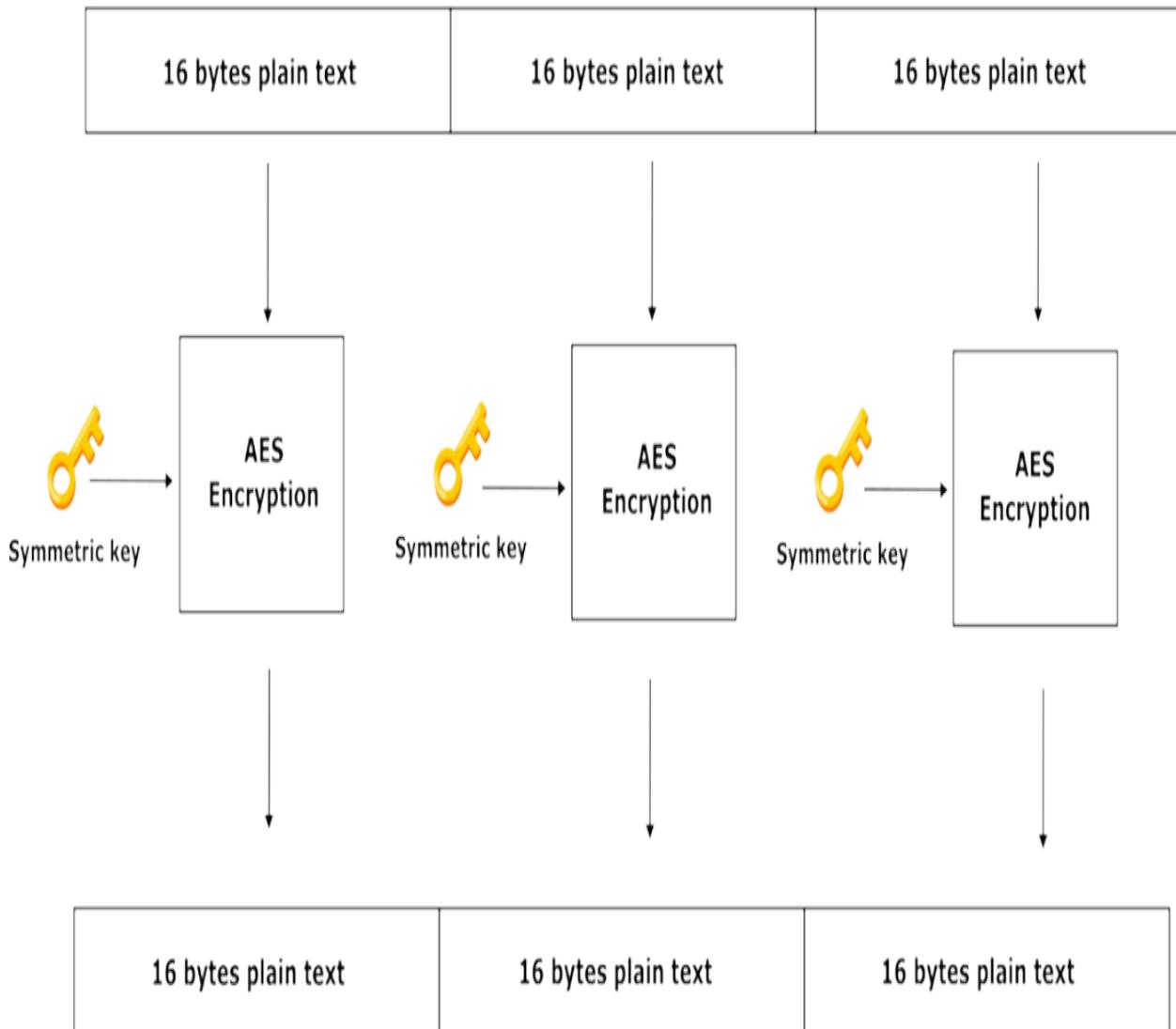
Definition

A mode of operation is a method that describes how to apply a block cipher algorithm to encrypt or decrypt data larger than a single block.

For example, in the *Electronic Code Book* (ECB) mode of operation, each 16-byte block is encrypted independently using the secret key, and then the cipher text is concatenated to produce the encrypted output (figure 5.3).

Figure 5.3 The Electronic Codebook Operating (ECB) Mode encrypt each input block independently using the secret key, concatenates the resulting cipher text. ECB mode is simple, but it is insecure, never use ECB mode.

Input of arbitrary length is broken up into blocks of 16 bytes (128-bits) so that each block can be encrypted. Additional padding is needed if the initial content cannot be exactly split and the last block is smaller.



If the input blocks repeat, then the cipher text will repeat because the AES encryption function is deterministic. Given the same input block and the same secret, it will always produce the same output. An attacker can count how many times a cipher text repeats giving the attacker information about repeating patterns which exist in the clear text.

For example, suppose Electronic Code Book (ECB) mode of operation is used to encrypt HTTP requests and responses. The attacker knows that every successful HTTP response starts with the line `HTTP/1.1 200 OK`, which is 15 characters. Adding the new line character yields a 16-byte block. AES is deterministic so encrypting `HTTP/1.1 200 OK` 100 times with the same key produces the same cipher text every time. The attacker looks for repeating patterns in the cipher text to deduce that a particular sequence of cipher text corresponds to `HTTP/1.1 200 OK`. The attacker can manipulate HTTP responses by substituting the cipher text that corresponds to `HTTP/1.1 200 OK` in places where the response code should not have been `200 OK`. The attacker can change the meaning of the response without ever needing to crack the encryption key. The ECB operating mode is part of the AES standard, but it is not secure and should never be used.

Warning

Never use ECB mode for anything in your applications. When searching online for code samples on implementing AES in your favorite programming language you will run into samples that use AES/ECB mode. **Don't just copy and paste code lest you use ECB mode accidentally.** Make sure you never ever use ECB mode unless you are reading historical data encrypted with AES ECB mode, and you need to decrypt it so you can re-encrypt it with a better AES mode.

Block cipher mode of operation is a generic concept that works with any encryption algorithm that works on fixed-sized input blocks. There are many modes of operation to choose from. AES supports the following modes of operation:

- *Electronic Code Book (ECB)* Encrypts each block of data independently, but is insecure because identical plaintext blocks produce identical ciphertext blocks.
- *Cipher Block Chaining (CBC)* Encrypts each block based on the previous block, making it more secure than ECB but requiring an initialization vector (IV).
- *Cipher Feedback (CFB)* Converts a block cipher into a stream cipher, allowing encryption of data smaller than the block size.

- *Output Feedback Mode (OFB)*: Similar to CFB but generates the keystream independently of the plaintext, making it resistant to error propagation.
- *Counter Mode (CTR)* Transforms a block cipher into a stream cipher by encrypting a counter for each block, offering high efficiency and parallelism.
- *Galois Counter Mode (GCM)* Combines CTR mode for encryption with Galois Field multiplication for authentication, ensuring both confidentiality and integrity.
- *Synthetic Initialization Vector (SIV)* Provides encryption and authentication while being resistant to IV misuse, ensuring security even if the IV is repeated.
- *AES-GCM-SIV* A variant of GCM that offers better protection against nonce reuse while maintaining high performance.

Applications always use AES in a specific mode of operation. Data encrypted with one mode of operation cannot be decrypted with another mode of operation. Each mode of operation makes a different set of security, usability, and performance tradeoffs.

TIP

If you get a statement such as “This application encrypts data using AES 256-bit key,” you must ask the question, “What AES operating mode is the application using?” Unless you know which mode is in use, you cannot evaluate the security of the encrypted data.

AES supports many operating modes because it is designed to be usable in various situations that demand different security, power consumption, and simplicity tradeoffs. For example, a low-power temperature sensor installed in an office building has very different security requirements than a publicly facing credit card processing API, but both can use AES in an operating mode that makes the right tradeoffs.

Unfortunately, some modes of operation, such as Electronic Code Book (ECB), are insecure, and you should never use them. Other modes are covered by patents, and so they are not widely deployed. Explaining all the AES operating modes is beyond the scope of this book. We will focus on two

commonly used modes *Cipher Block Chaining* (CBC) and *Galois Counter Mode* (GCM) because these are the modes you are most likely to encounter in enterprise applications.

TIP

If you are unsure what AES mode to use and your company does not have a published recommendation, go with Galois Counter Mode (GCM) or one of its variations, such as AES-GCM-SIV.

5.2.1 Cipher Block Chaining (CBC) mode

Cipher Block Chaining (CBC) is an operating mode for AES encryption designed to make each block of ciphertext appear random, even when the same plaintext is encrypted multiple times. This helps prevent attackers from spotting patterns in the ciphertext that could reveal clues about the original data. For example, if a message contains repeated sections, CBC ensures these repetitions are not visible in the encrypted output.

CBC achieves this by "chaining" blocks of data together during encryption. Each plaintext block is combined with the ciphertext of the previous block before being encrypted. This process ensures that the encryption of one block depends on the encryption of the block before it, making patterns impossible to detect.

To use AES in CBC mode, you need two essential components:

- *A Secret Key*: This is used to perform the encryption and decryption. It must be shared securely between the sender and receiver.
- *An Initialization Vector (IV)*: This is a random value used to kickstart the encryption process for the first block, ensuring the encryption is unique each time, even if the same plaintext and key are used.

The secret key is only known to the parties that should have access to the data. The *Initialization Vector* (IV) is a random 128-bit value that is used in CBC mode to ensure that the generated cipher text does not have any repeating patterns even if the clear text being encrypted has repeating patterns. The word vector might remind you of college mathematics. Rest

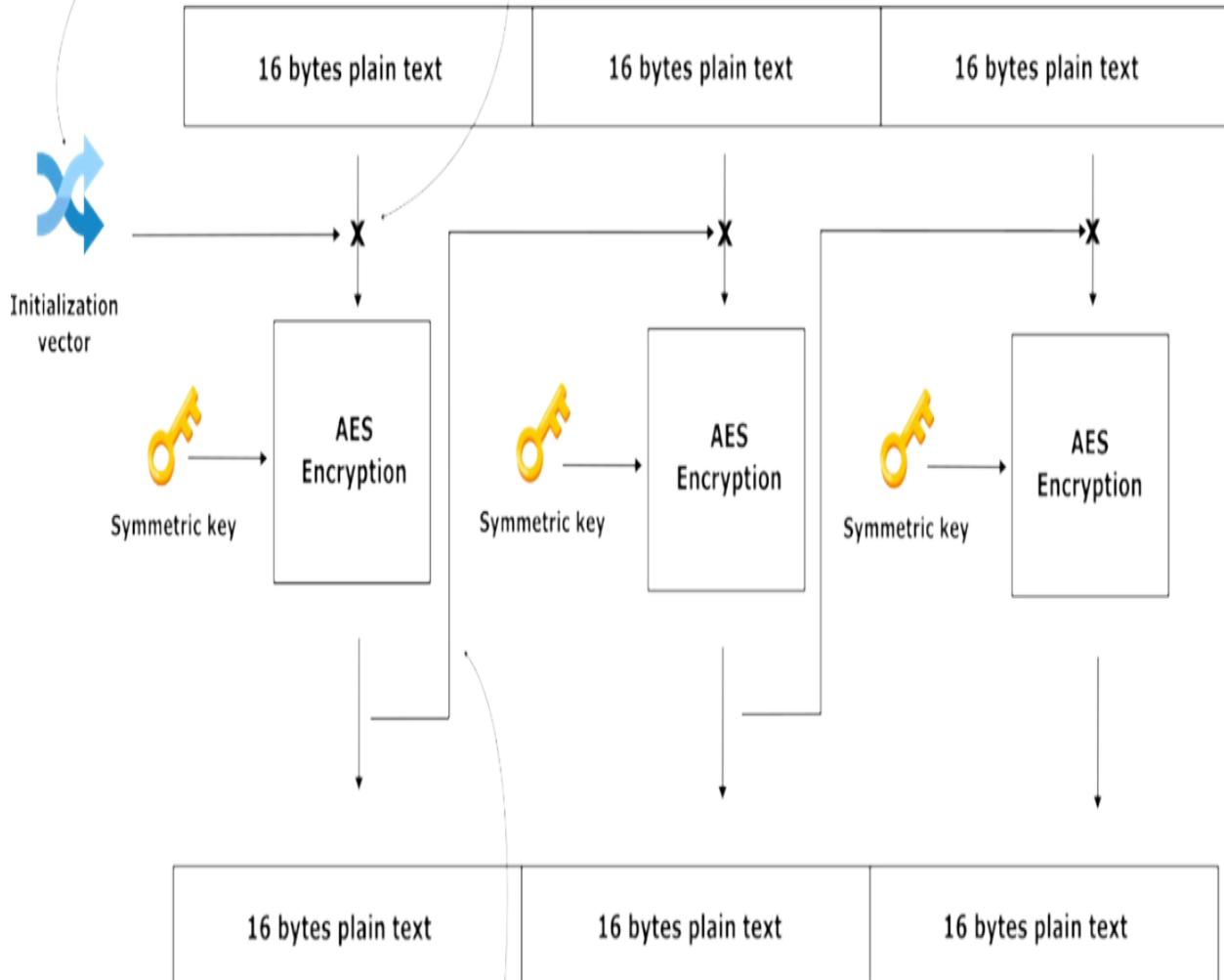
assured, in the context of AES initialization, a vector is just a random value that should only be used once.

In cryptography terminology a number that is only used once is called a *nonce*. The diagram in figure 5.4 provides a visual explanation of CBC operating mode.

Figure 5.4 The symbol with + inside a circle is the XOR operation. Cipher Block Chaining (CBC) takes the output of one block and feeds it into the next block. The net result is that even if you encrypt the same data multiple times the cipher text looks random. An attacker cannot determine any patterns in the underlying data by looking for patterns in the cipher text.

Randomly generated 128-bit value mixed into the input being encrypted.

The initialization vector is mixed with the clear text to ensure that repeating patterns in the plain text are removed before encryption.



The output of the first block is used as vector for the next block. It is mixed with the next block to ensure repeating patterns are eliminated before the encryption.

The initialization vector (IV) is combined with the first 128-bit plain text block to generate the first cipher block. The first cipher block is then

combined with the second plain text block with the result encrypted using AES, this chaining process repeats for all the blocks.

To decrypt data encrypted with AES CBC mode both the key and initialization vector are required. Only the key is treated as a secret value, the initialization vector is stored as cleartext so that the cipher text can be decrypted later. The initialization vector should only ever be used once with a specific key.

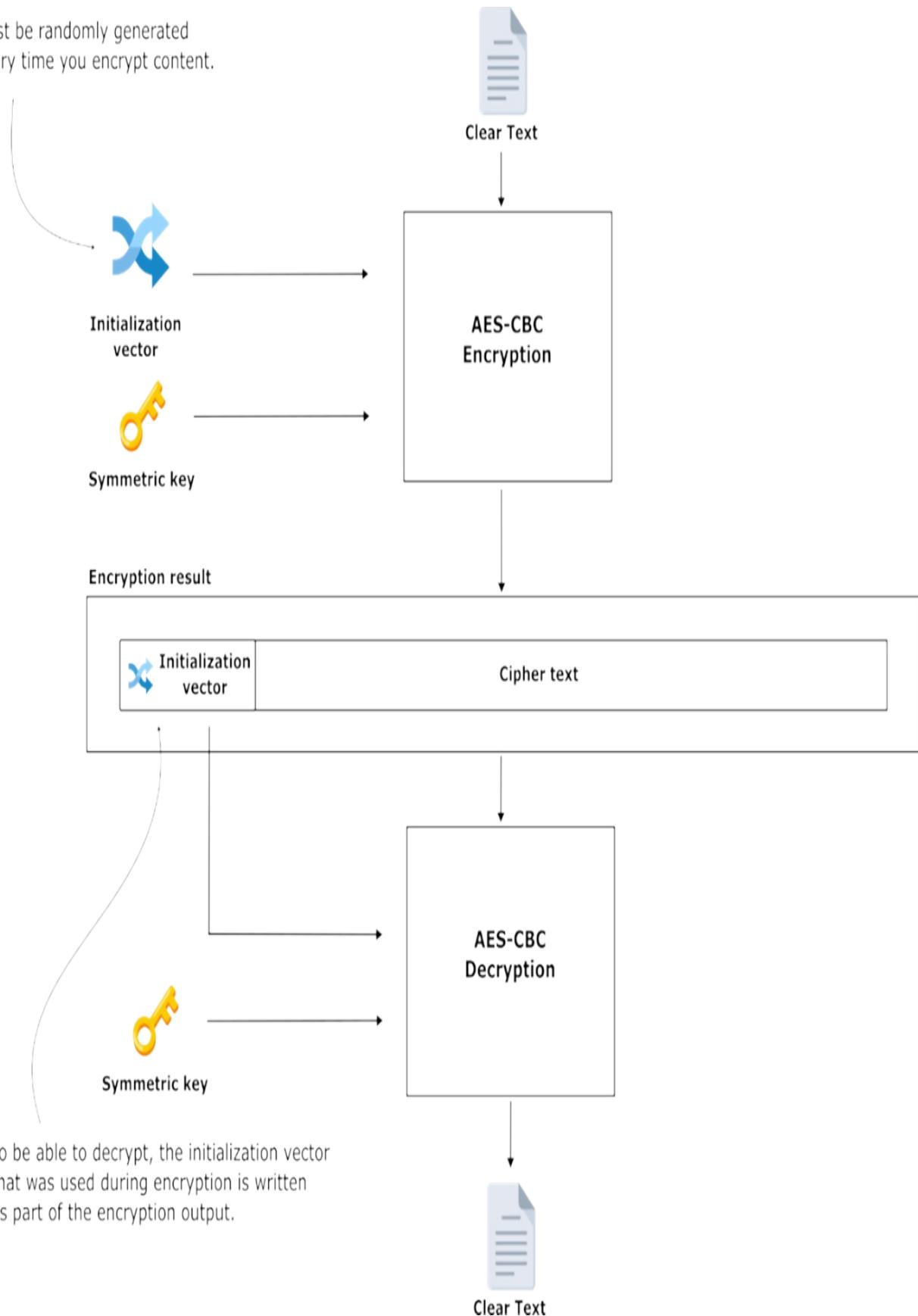
Note

If you want to encrypt two different files with the same AES key, you must use a different initialization vector for each encryption operation.

A good mental model for understanding AES in CBC mode is to think of a black box that takes an input message of any length, a secret key, and a random initialization vector. The purpose is to produce an output consisting of the initialization vector, followed by the cipher text. Figure 5.5 provides a visualization of the AES in CBC mode.

Figure 5.5 AES in CBC mode can be visualized as a black box that takes three inputs: plain text, key, and initialization vector. The initialization vector is written out as a plain text to output followed by the cipher text computed by the AES in CBC mode. Decryption process requires three inputs: initialization vector, cipher text, and key. The initialization vector and cipher text can be read from the output bytes, while the key must be provided by the party performing the decryption.

Must be randomly generated every time you encrypt content.



Warning

If a key and initialization vector combination is reused you can end up with catastrophic encryption failure. For example, an attacker might be able to compute the original key used to perform the encryption or recover the cipher text. The exact impact of reusing an initialization vector depends on the mathematics underlying the operating mode. It is beyond the scope of this book to delve into the detailed implications. The key takeaway is that you should never reuse an initialization vector across encryption operations.

TIP

Every time you call the AES encryption function you should generate a brand new random initialization vector using a cryptographically secure random number generator such as `java.security.SecureRandom`.

Another interesting thing to consider about AES-CBC is that if a file is encrypted with AES-CBC, an attacker can tamper with it by randomly modifying some bits. Upon decryption, AES-CBC will return corrupt plain text, causing trouble for the user of the decrypted plain text.

In a blog post[\[1\]](#) Andrew Tierney demonstrates how an attacker with access to the AES-CBC encrypted cipher text for the message “A dog’s breakfast” can modify the message to say a “A cat’s breakfast” without knowing the encryption key or breaking the AES algorithm. AES-CBC mode guarantees the confidentiality of the encrypted data (Table 5.1). It does not protect the integrity or the authenticity of the data. Table 5.1 shows security goals can be realized by AES-CBC operating mode.

Table 5.1 Cryptography goals and algorithms matrix

Goal	SHA-2	SHA-3	HMAC	AES-CBC
Integrity	Yes	Yes	Yes	No
Authentication	No	No	Yes	No
Confidentiality	No	No	No	Yes
Non-repudiation	No	No	No	No

WARNING

AES-CBC mode provides confidentiality only. Real-world security requires integrity, authentication, and confidentiality. Avoid using AES-CBC mode. Use AES-GCM mode, which we will explain shortly.

5.2.2 Authenticated encryption

As we saw in the previous section, data confidentiality alone is insufficient; an attacker can tamper with encrypted data in highly determinantal ways to consume applications. Authenticated encryption refers to any encryption scheme that provides data integrity, authenticity, and confidentiality.

Authenticated encryption can be implemented by combining Message Authentication Codes (MACs) with an encryption algorithm. For example, data is encrypted with AES-CBC mode, and then an HMAC is computed on the cipher text. Before decryption, the HMAC of the cipher text is used to ensure that the cipher text was not tampered with. There are three possible ways to combine encryption and MACs.

- *Encrypt-and-Mac* - where the plain text is encrypted, and the MAC of the plain text is calculated this approach is used by the Secure Shell (SSH) protocol. Encrypt-and-Mac should only be used by cryptography experts since it can lead to subtle security bugs.
- *Mac-then-Encrypt* - first computes the MAC on the plain text and then encrypts both the plaintext and the Mac. This approach is used by the Transport Layer Security (TLS) 1.2 protocol. Only cryptography experts should use Mac-then-Encrypt since it can lead to subtle security bugs.
- *Encrypt-then-Mac* - first encrypt the plain text, then compute the MAC of the encrypted text, this approach has been used by the IP Security (IPSec) protocol. Encrypt-then-Mac is the most secure approach for combining encryption and message authentication codes.

Combining encryption and MACs correctly is tricky. Cryptographers designed several modes of operation for AES that provide authenticated encryption. Galois Counter Mode (GCM) is one of the most widely used authentication encryption modes and is the subject of the next section.

WARNING

The designers of major protocols such as SSH and TLS made mistakes in combining encryption with message authentication codes, causing vulnerabilities in early versions of these protocols. As an application developer, use an authenticated encryption operating mode such as Galois Counter Mode (GCM) instead of rolling your own authenticated encryption scheme

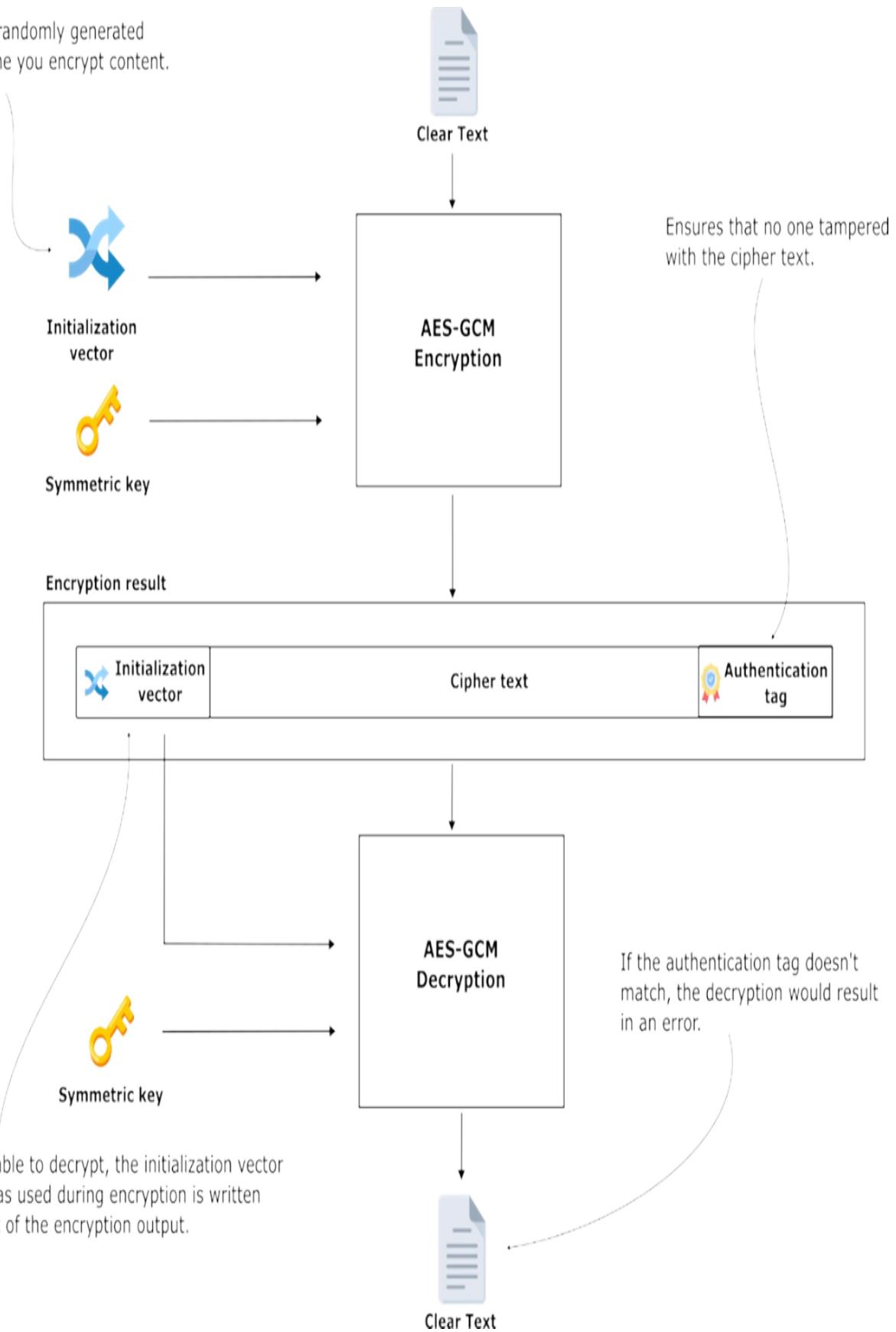
5.2.3 Galois Counter Mode (GCM)

AES *Galois Counter Mode* (GCM) is an operating mode that ensures integrity, authenticity, and confidentiality. AES-GCM provides the equivalent security of first encrypting with AES-CBC mode and then computing hashed message authentication code over the cipher text. Explaining the mathematical details of how GCM mode works is beyond the scope of this book and we'll focus instead on essential details any developer should know.

Think of AES-GCM mode as a black box that takes three inputs: plain text, a secret key, and a random initialization vector, to produce an output consisting of the initialization vector, followed by the cipher text and a message authentication code. Figure 5.6 provides a visualization of the AES in GCM.

Figure 5.6 AES in GCM mode as a black box that processes three inputs: plaintext, key, and initialization vector. The algorithm outputs the initialization vector (written in plaintext), followed by the ciphertext it computes, and finally the authentication code generated by GCM mode. To decrypt, you provide four inputs: the initialization vector, ciphertext, key, and authentication code. The algorithm uses these inputs to verify the data's integrity and recover the original plaintext.

Must be randomly generated every time you encrypt content.



In GCM mode, the decryption algorithm first checks that the authentication code is valid. If the code is invalid, an error is raised, and decryption is not attempted. As an application developer, you should always use an authenticated encryption operating mode. AES-GCM is a widely implemented authenticated encryption mode. Therefore, GCM should be your default mode whenever you use AES. AES in GCM mode delivers three of the four goals of cryptography (table 5.2).

Table 5.2 Cryptography goals and algorithms matrix

Goal	SHA-2	SHA-3	HMAC	AES-CBC	AES-GCM
Integrity	Yes	Yes	Yes	No	Yes
Authentication	No	No	Yes	No	Yes
Confidentiality	No	No	No	Yes	Yes
Non-repudiation	No	No	No	No	No

Warning

Reusing an initialization vector with GCM is catastrophic. Never re-use initialization vectors with AES. Initialization vectors must also be random. While the AES algorithm is very secure, it is easy to make mistakes when writing code that uses AES, for example, re-using an initialization vector, not using enough randomness, or any other ways of incorrectly coding. Secure coding practices and code reviews ensure you don't accidentally introduce a vulnerability.

5.2.4 Exercises

1. What extra guarantees does AES-GCM provide compared to AES-CBC?
2. Why is reusing an initialization vector (IV) in AES-GCM dangerous?
3. Which cryptographic goals are fulfilled by AES-CBC vs. AES-GCM?
4. In the ACME refunds scenario, what happens if the encrypted refunds.json file is tampered with?

5.3 Java Support for AES

Java 21 ships with first-class support for AES in several operating modes, including CBC and GCM. Working with AES requires generating initialization vectors, which are random sequences of bytes. The utility method in listing 5.1 generates the requested number of random bytes using a cryptographically secure random number generator. You find this method in project ssfd_ch5_ex1 provided with the book.

Listing 5.1 Generating Secure Random Numbers

```
import java.security.SecureRandom;

public class CryptoUtils {

    private static final SecureRandom secureRandom
        = new SecureRandom(); #A

    private static byte[] randomBytes(int length) { #B
        byte[] bytes = new byte[amount];
        secureRandom.nextBytes(bytes);
        return bytes;
    }
}
```

Warning

in Java, the `java.util.Random` is not cryptographically secure and should never be used when generating random bytes for cryptographic applications. Always use the `java.security.SecureRandom` for cryptographic applications. When coding in an IDE be careful not to import `Random` instead of `SecureRandom` accidentally.

Listing 5.3 shows a utility function that takes an array of bytes and encrypts using AES in GCM operating mode with a 256-bit key. You find this method in the project ssfd_ch5_ex2 provided with the book.

Listing 5.2 Encrypting with AES/GCM

```
public static byte[] encryptAes256GCM(
    byte[] clearText,
```

```

byte[] key) {

try {
    byte[] iv = generateRandomBytes(12); #A

    Cipher cipher = Cipher
➥.getInstance("AES/GCM/NoPadding"); #B

    GCMParameterSpec gcmSpec =
        new GCMParameterSpec(128, iv);

    SecretKeySpec keySpec = new SecretKeySpec(key, "AES");

    cipher.init(Cipher.ENCRYPT_MODE,
➥ keySpec, gcmSpec); #C

    byte[] cipherText = cipher.doFinal(clearText); #D

    byte[] result = new byte[iv.length
➥+ cipherText.length]; #E

    System.arraycopy(iv, 0, result, 0, iv.length);
    System.arraycopy(cipherText, 0, result, iv.length, cipherText

    return result;

} catch (Exception e) {
    throw new RuntimeException("AES-256-GCM encryption failed", e
}
}

```

Code in listing 5.2 works by first obtaining an instance of `javax.crypto.Cipher` using the `Cipher.getInstance()` method. `Cipher` is the JCE API for encrypting and decrypting data, it is a stateful object that is not *thread safe*.

The algorithm name has the pattern “algorithm/mode/padding” for example “AES/GCM/NoPadding” the “Java Security Standard Algorithm Names”[\[2\]](#) contains a list of all the algorithm names that can be passed to `Cipher.getInstance()` method. Before the cipher can be used it must be initialized with three parameters.

1. The mode to configure the cipher in encrypt or decrypt mode

2. The AES key to use
3. The GCM configuration in this case 128-bit message authentication tag and 12-byte initialization vector. 12-byte is the recommended NIST size for an initialization vector but always check with your info sec team for important algorithm choices such as initialization vector.

The cipher can be called with all the data to be encrypted using the `doFinal()` method. Alternatively, you can call the `update()` method and pass the data in chunks, which can be useful when encrypting a large file that you don't want to read into a byte array. The `Cipher` class defines methods for processing associated data. Please consult the Javadoc for the `Cipher` class for the full details.

The initialization vector must be available to the code that decrypts the cipher text. The above sample concatenates the 12-byte initialization vector with the cipher text. The decryption code (listing 5.3) will need to break the input byte array into two parts the 12-byte initialization vector and the cipher text as show in the listing below.

Listing 5.3 Decryption of AES/GCM encrypted data

```
public static byte[] decryptAes256GCM(byte[] ivAndCiphertext, byte[]
    try {
        byte[] iv = new byte[12]; #A
        byte[] cipherText
    ↵ = new byte[ivAndCiphertext.length - 12]; #A

        System.arraycopy(ivAndCiphertext, 0, iv, 0, 12); #A
        System.arraycopy(ivAndCiphertext, 12, #A
            cipherText, 0, cipherText.length);

        Cipher cipher
    ↵ = Cipher.getInstance("AES/GCM/NoPadding"); #B
        GCMParameterSpec gcmSpec
    ↵ = new GCMParameterSpec(128, iv); #B
        SecretKeySpec keySpec
    ↵ = new SecretKeySpec(key, "AES"); #B
        cipher.init(Cipher.DECRYPT_MODE,
    ↵ keySpec, gcmSpec); #B

        return cipher.doFinal(cipherText); #C
    } catch (Exception e) {
```

```
        throw new RuntimeException("AES-256-GCM decryption failed", e
    }
}
```

TIP

As we discussed also in chapter 4, you can use alternative libraries that sometimes make the code cleaner and simpler. An alternative that we also discussed in chapter 4 is Google Tink. You can find Tink at <https://github.com/google/tink>.

Listing 5.4 shows you how to use Google Tink to encrypt and decrypt with AES in GCM mode. As you can see, we are just using the methods defined by the library and we didn't need to implement ourselves methods that do the operations. You find this example in project ssfd_ch5_ex3 provided with the book.

Listing 5.4 Encryption and decryption with AES in GCM mode using Google Tink

```
public static void main(String[] args) throws Exception {
    AeadConfig.register();

    KeysetHandle keysetHandle = KeysetHandle.generateNew(
        AesGcmKeyManager.aes256GcmTemplate());

    Aead aead = keysetHandle.getPrimitive(Aead.class);

    byte[] plaintext = "Hello, Tink AES-GCM!"  

        .getBytes(StandardCharsets.UTF_8);

    byte[] ciphertext  

    ↪ = aead.encrypt(plaintext, null); #A

    byte[] decrypted  

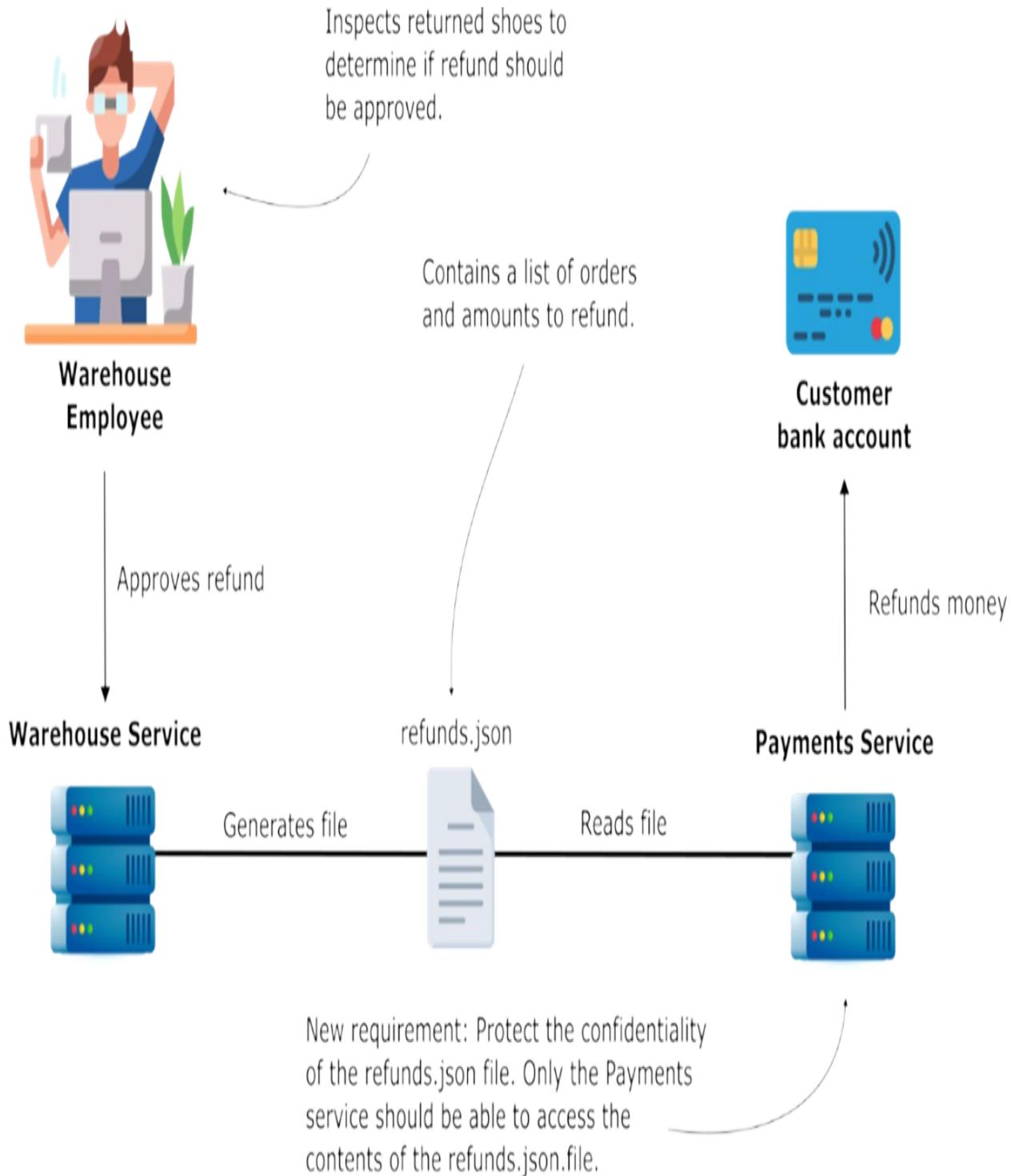
    ↪= aead.decrypt(ciphertext, null); #B

    // Print results
}
```

5.3.1 The ACME Inc. Scenario

Recall the ACME Inc. online shoe retailer refund processing scenario we used in the previous chapter. Customers mail shoes they do not like to ACME Inc's warehouse. Warehouse employees verify that the returned shoes are in good condition before authorizing a credit card refund using the warehouse management app. Once a day a `refunds.json` file is produced by the warehouse management application and sent to the payments service to refund customer credit cards (figure 5.7).

Figure 5.7 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a `refunds.json` file. The payment service refunds customer credit cards for the amount specified in the `refunds.json` file.



File contents are described by the next snippet. We used the same content scheme in chapter 4 where we discussed hashing functions, message authentication codes (MACs) and Hash Message Authentication Codes (HMACs).

```
[ {  
    "orderId" : "12345",  
    "amount" : 500  
, {  
    "orderId" : "56789",  
    "amount" : 250  
} ]
```

In chapter 4 we used *Hash Message Authentication Code* (HMAC) to detect data corruption or tampering of the `refunds.json` file. In this chapter, we want to protect the integrity, authenticity, and confidentiality of the `refunds.json` file.

5.3.2 Implementing the ACME Inc. Scenario

In the ACME Inc. scenario, the warehouse service sends a `refunds.json` file to the payments service so that it can issue refunds to customer credit cards.

To guarantee integrity, authenticity and confidentiality of the `refunds.json` file the Warehouse service (project `ssfd_ch5_ex4-warehouse` provided with the book) encrypts it using AES-GCM mode with a 256-bit key.

The Payments service (project `ssfd_ch5_ex4-payments`) decrypts `refunds.json` using AES-GCM which throw an error if the cipher text has been tampered with or the key is wrong. Since the secret key is only known to the warehouse and the payments service the payment service can assume that the warehouse service created the `refunds.json`.

Listing 5.5 show you the simple implementation of the service class in the Payments service. The service decrypts the data with AES-GCM using the secret which is configured in the `application.properties` file. The plain text result is put as response body. I might repeat myself throughout the book, but it's important: Never configure secrets directly in the properties files or hardcoded in a real-world project! Never! In a real-world scenario, all the secret details must be securely stored in a security vault (such as Hashicorp - <https://www.hashicorp.com>).

Listing 5.5 The RefundService class decrypts the data

```

@Service
@RequiredArgsConstructor
public class RefundService {

    private final ObjectMapper objectMapper;
    private final AesGcmManager aesGcmManager;

    public List<Refund> decryptAndReturnRefunds(byte[] encryptedBodyBytes) {
        byte[] plainBytes = aesGcmManager
            .decrypt(encryptedBodyBytes); //A
        try {
            return objectMapper.readValue(plainBytes,
                new TypeReference<List<Refund>>(){});
        } catch (IOException e) {
            throw new IllegalArgumentException("Invalid " +
                "refunds JSON format", e);
        }
    }
}

```

The steps to test this example are:

1. Run the Warehouse application to generate the refunds.json.aesgcm file which contains the AES-GCM encryption of the refunds.json file content.
2. Run the Payments service which is a Spring Boot application exposing an endpoint with decrypts the content and puts the plain text back on the response.
3. Call the endpoint the Payments service exposes to prove the decryption works. Make different tests to make sure you understood the example:
 - a. Change the key (in the application.properties file) and observe that the decryption doesn't work anymore.
 - b. Alter the content of the refunds.json.aesgcm and observe that the endpoint fails to decrypt the data.

You find detailed instructions about running the example in the README.md file of the Payments project.

With AES in GCM mode, any corruption in the file will trigger an error during decryption, ensuring the issue is detected. In contrast, with CBC mode, the decryption process would still complete, but the resulting file

would be corrupted, and the error might go unnoticed.

5.4 Authenticated Encryption with Associated Data (AEAD)

Consider a scenario where a message must be sent securely between two systems. The message has a header containing metadata, which intermediary systems use to route the message to its final destination. The message body contains content that must be kept confidential, so it is encrypted (figure 5.8). The intermediary systems don't have access to the message encryption key, so the header must be plain text so that the intermediary systems can read it and route the message.

Figure 5.8 A message with a plain text header and an encrypted body is very common.

The header must be clear text to allow intermediary systems to read the header values.

The body must be encrypted to ensure confidentiality.



We must be able to detect any tampering with either the header or the body.

In some situations, attackers can manipulate messages in ways that could disrupt an application:

- *Tampering with the Header:* An attacker could intercept the message and modify its header, potentially routing it to the wrong recipient.
- *Swapping Message Bodies:* An attacker might intercept two messages, swap their encrypted bodies, and send them to the original recipients, causing confusion or errors in the receiving applications.

To prevent these types of attacks, the receiving application must be able to detect if the header has been altered or if the encrypted body has been

swapped. In security terminology, the header is called Additional Associated Data (AAD). This term means that the header is plaintext data that is associated with, and bound to, the encrypted portion (ciphertext) of the message. Ensuring this association is crucial for security.

A common approach is to use a message authentication code (MAC) to ensure the integrity of both the header and the body. This involves:

- Encrypting the body using AES.
- Creating a MAC over the combined header and encrypted body, often using HMAC.

While this method is effective, it requires two separate algorithms—HMAC for the MAC and AES for encryption—which can add complexity.

As mentioned earlier, an authenticated encryption mode combines both encryption and MAC creation into a single process. This mode type is known as Authenticated Encryption with Associated Data (AEAD). AEAD allows you to:

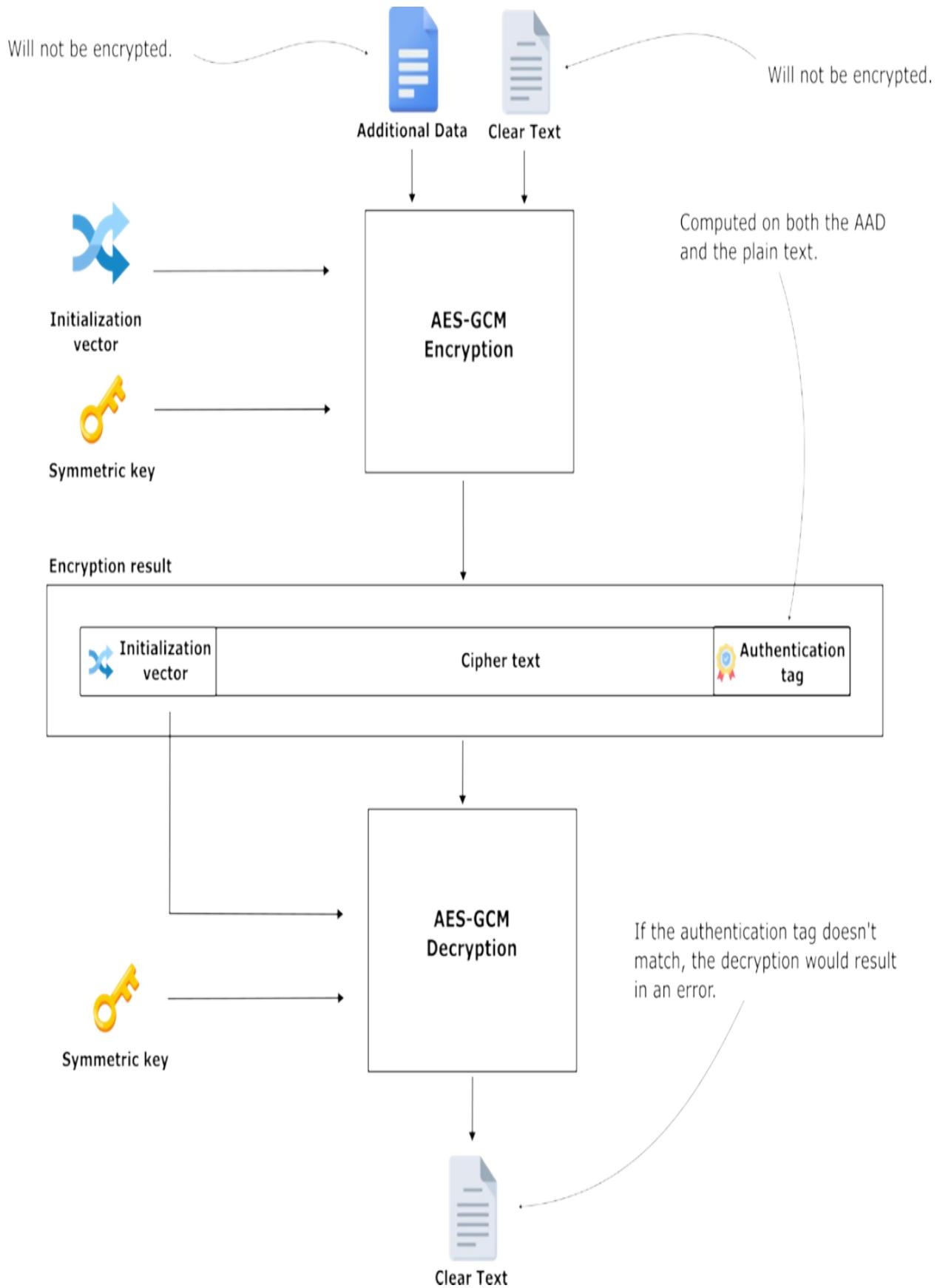
- Leave some data (like the header) as plaintext.
- Encrypt the rest of the data (like the body).
- Compute a MAC over both the plaintext (header) and the ciphertext (body) to ensure they are tightly bound and have not been tampered with.

Essentially, AEAD achieves the same result as performing $\text{HMAC}(\text{header} \parallel \text{AES}(\text{body}))$, but with just one algorithm.

AES in Galois Counter Mode (GCM) is one of the most popular encryption modes because it supports AEAD. It simplifies the process by combining encryption and authentication into a single operation (figure 5.9), making securing messages easier and more efficient while protecting against tampering and swapping attacks.

Figure 5.9 AES in GCM mode can be visualized as a black box that takes four inputs: plain text, secret key, initialization vector, and optional Additional Associated Data (AAD). The authentication tag is computed using the AAD data and the cipher text. The initialization vector, cipher text, and authentication tag are saved to the output. To decrypt you must provide both the

encryption key, and the AAD data otherwise an error is raised.



AEAD is useful outside the context of message in transit. Consider a payment API storing information about transactions in a SQL database. The transactions table has the columns transaction ID, credit card number, amount, and date.

To comply with security standards, we use AES to encrypt the credit card number before we store it in the database. A malicious DBA, a hacker, or a buggy code can take the encrypted credit card number from one row and swap it with the encrypted credit card number in another row. The hacker does not need to decrypt the credit card number. Just cause havoc by corrupting the database. By running AES in GCM mode, we can encrypt the credit card number and compute an authentication code that factors in the transaction ID, amount, and date; this way, we can detect if the data in a row has been tampered with. Some databases, such as Google Big Query, have native support for AEAD[\[3\]](#).

Listing 5.6 shows a simple example of encryption with AES in GCM mode using AEAD. You find this example in project ssfd_ch5_ex5 provided with the book.

Listing 5.6 Encryption with AEAD with AES-GCM in Java

```
public static byte[] encryptAes256GCM(byte[] clearText,
→byte[] key,
→byte[] aad) { #A
try {
    byte[] iv = generateRandomBytes(12);

    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    GCMPParameterSpec gcmSpec = new GCMPParameterSpec(128, iv);

    SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
    cipher.init(Cipher.ENCRYPT_MODE, keySpec, gcmSpec);

    if (aad != null && aad.length > 0) { #B
        cipher.updateAAD(aad);
    }

    byte[] cipherText = cipher.doFinal(clearText);

    byte[] result = new byte[iv.length + cipherText.length];
```

```

        System.arraycopy(iv, 0, result, 0, iv.length);
        System.arraycopy(cipherText, 0, result, iv.length, cipherText
            return result;
    } catch (Exception e) {
        throw new RuntimeException("AES-256-GCM encryption failed",
    }
}

```

Listing 5.7 shows the decryption for ciphers with AAD.

Listing 5.7 Decryption with AEAD with AES-GCM in Java

```

public static byte[] decryptAes256GCM(byte[] ivAndCiphertext,
                                         byte[] key,
                                         byte[] aad) {
    try {
        byte[] iv = new byte[12];
        byte[] cipherText = new byte[ivAndCiphertext.length - 12];
        System.arraycopy(ivAndCiphertext, 0, iv, 0, 12);
        System.arraycopy(ivAndCiphertext, 12,
                        cipherText, 0, cipherText.length);

        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec gcmSpec = new GCMParameterSpec(128, iv);
        SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
        cipher.init(Cipher.DECRYPT_MODE, keySpec, gcmSpec);

        if (aad != null && aad.length > 0) { #A
            cipher.updateAAD(aad);
        }

        return cipher.doFinal(cipherText);
    } catch (Exception e) {
        throw new RuntimeException("AES-256-GCM decryption failed"
    }
}

```

Compressing the data

Compressing data before moving over the network or storing it can reduce costs and improve performance. Frequently, we want to combine encryption and compression to meet security, performance, and cost requirements. Therefore, as a developer, you need to make a choice. Should data be compressed first and then encrypted, or should we encrypt the data first and

then compress it?

Compression algorithms work by replacing frequently repeating patterns of data with shorter patterns. Encryption algorithms provide security by making the output look like a random data stream without repeating patterns. Therefore, compression algorithms perform poorly against encrypted cipher text. It is better to compress data first and then encrypt it so that you get the maximum compression ratios.

5.4.1 Exercises

5. Why should compression be applied before encryption, not after?
6. What happens to compression ratios if you try to compress already encrypted data?
7. In a system where messages are both compressed and encrypted, what risk exists if encryption is applied before compression?

5.5 AES Best Practices

We have covered a lot of details about AES so far. As you have noticed from the various warning boxes in this chapter, it is easy to use AES incorrectly and thus build an insecure system. Without professional advice from a security specialist for your AES, you should opt for using AES-GCM with a 256-bit key. In the following sections we discuss more details about this recommendation.

5.5.1 Selecting the AES key size

The AES secret key can be 128, 192, or 256 bits long. Which key size is best? An attacker can write a program that tries out every possible key combination in hopes of finding the correct encryption key. Trying out every secret key combination is called a brute force attack.

Using an electronic computer to brute force a 128-bit AES key requires quadrillions of years of computing time. A quadrillion is 1 million billion years. The sun will explode in 5 billion years, so there is no practical way to brute force a 128-bit AES key using electronic computers available at the

time of writing. But what if you have a quantum computer?

Quantum computers are a new type of computer that can perform computations that are not feasible on a classical electronic computer. In theory, a quantum computer running Grover's algorithm can speed up the brute force attack against AES. With a quantum computer, a 128-bit AES key has the same strength as a 64-bit key, and a 256-bit key has the strength of a 128-bit key.

The theory of quantum computing is well understood; however, building a practical quantum computer is an unsolved engineering challenge at the time of writing. Cryptographers are building encryption algorithms that can resist quantum computers, but no algorithm has been standardized so far.

The AES algorithm is considered mathematically secure. However, in real life you will need to use an AES implementation. An AES implementation may contain bugs that weaken security. The AES encryption key must be stored somewhere, so hackers look for ways to steal the encryption key rather than trying to mathematically break AES.

There have been many cases of developers publishing secret keys to GitHub^[4] repos accidentally or through ignorance. A 256-bit AES key will not keep data secure if the attacker steals the key. Managing keys securely is both a human and technical problem. This book will show you how to use HashiCorp Vault and public cloud key management services to store encryption keys. However human processes for handling secret keys are beyond the scope of this book.

TIP

Use 256-bit AES keys.

5.5.2 Checklist for using AES-GCM correctly in Java

The checklist below provides you with best practices to follow as a developer to use AES correctly in GCM mode.

1. Use a 256-bit key for optimal security.

2. Generate the key with a cryptographically secure random number generator such as `java.security.SecureRandom`.
3. Keep the key secret, store it in a secure key storage system and follow secure process for handling key materials.
4. Use a 96-bit initialization vector for AES-GCM, a longer initialization vector will be shortened by AES-GCM to be 96-bits long.
5. Generate the initialization vector using a cryptographically secure random number generator such as `java.security.SecureRandom`.
6. Never ever reuse an initialization vector since reuse of (key, initialization vector) combination with AES-GCM can result in a catastrophic security failure. Always generate a new random initialization vector before calling the `cipher.init()` method in java.
7. Use a 128-bit tag size configured via the `javax.crypto.spec.GCMParameterSpec` class in java. 128-bit is the largest authentication tag size possible with AES-GCM.
8. The maximum amount of data that can be encrypted by a key, and initialization vector combination is approximately 68GB or $(2^{39} - 16)$ bytes in size. If you need to encrypt more than 68GB with a single AES key, initialization vector combination you will need to use a different AES mode or split up the data being encrypted into 68GB chunks.

TIP

This list above is challenging to follow in your application code. Use a library that implements the above best practices and offers a developer-friendly API that is hard to accidentally misuse. Google Tink is a good choice as it has implementations in Java, C++, Objective-C, Go, Python, and JavaScript and is maintained by the Google security team. Chapter 7 covers Google Tink. You can find Tink at <https://github.com/google/tink>.

5.5.3 Exercises

8. What is the recommended AES key size for strong security today?
9. How does a quantum computer change the effective strength of AES keys?
10. Why is it dangerous to store AES keys directly in source code or config files?

11. What initialization vector (IV) size and policy should be used with AES-GCM?
12. Why is it better to use a library like Google Tink instead of hand-writing AES-GCM code in production?

5.6 Answers to exercises

1. What extra guarantees does AES-GCM provide compared to AES-CBC?
AES-GCM guarantees integrity, authenticity, and confidentiality, while AES-CBC only guarantees confidentiality.
2. Why is reusing an initialization vector (IV) in AES-GCM dangerous?
Reusing an IV with AES-GCM is catastrophic — it can allow attackers to recover the key or forge messages.
3. Which cryptographic goals are fulfilled by AES-CBC vs. AES-GCM?
AES-CBC: confidentiality only. AES-GCM: confidentiality, integrity, and authentication. Neither provides non-repudiation.
4. In the ACME refunds scenario, what happens if the encrypted refunds.json file is tampered with?
The Payments service will detect the tampering during decryption (authentication tag mismatch) and throw an error. No refunds will be processed.
5. Why should compression be applied before encryption, not after?
Because encryption produces random-looking data with no patterns, making it nearly impossible for compression algorithms to reduce size effectively.
6. What happens to compression ratios if you try to compress already encrypted data?
Compression ratios drop dramatically (often no size reduction at all) because ciphertext appears random and has no repeating sequences.
7. In a system where messages are both compressed and encrypted, what risk exists if encryption is applied before compression?
Compression after encryption wastes CPU cycles and may lead to inefficient or misleading system design, since the data won't shrink.
8. What is the recommended AES key size for strong security today?
256-bit keys are recommended for long-term security.
9. How does a quantum computer change the effective strength of AES

keys?

Quantum computers running Grover's algorithm halve the effective strength of AES keys (128-bit becomes like 64-bit; 256-bit becomes like 128-bit).

10. Why is it dangerous to store AES keys directly in source code or config files?

Because if attackers gain access to your code repository or config files, they can steal the key and decrypt all your data.

11. What initialization vector (IV) size and policy should be used with AES-GCM?

A 96-bit (12-byte) IV should be generated fresh with a cryptographically secure RNG for every encryption operation. Reuse of an IV is catastrophic.

12. Why is it better to use a library like Google Tink instead of hand-writing AES-GCM code in production?

Because libraries like Google Tink implement best practices automatically, reducing the risk of developer mistakes that introduce security flaws.

5.7 Summary

- Advanced Encryption Standard (AES) is the most widely used symmetric encryption algorithm, supported by all major cloud providers and operating systems for protecting data confidentiality.
- AES is a block cipher that encrypts fixed 128-bit blocks and requires a mode of operation to handle data larger than one block.
- Electronic Code Book (ECB) mode is insecure because identical plaintext blocks produce identical ciphertext blocks, revealing patterns to attackers.
- Cipher Block Chaining (CBC) mode chains blocks together using initialization vectors to prevent pattern detection but only provides confidentiality, not integrity or authenticity.
- Initialization vectors must be random, unique for each encryption operation, and never reused with the same key to prevent catastrophic security failures.
- Authenticated encryption combines encryption with message authentication to provide confidentiality, integrity, and authenticity in a

single operation.

- Galois Counter Mode (GCM) is the recommended AES mode because it provides authenticated encryption, ensuring confidentiality, integrity, and authenticity simultaneously.
- AES-GCM requires a secret key, random initialization vector, and produces ciphertext with an authentication tag that detects any tampering.
- Authenticated Encryption with Associated Data (AEAD) allows plaintext headers to be authenticated along with encrypted message bodies without encrypting the headers.
- Java provides AES support through the Cipher class, but developers should use cryptographically secure random number generators and follow specific configuration requirements.
- Best practices include using 256-bit keys, 96-bit initialization vectors, 128-bit authentication tags, and never reusing initialization vectors.
- Compression should be applied before encryption because encrypted data appears random and cannot be effectively compressed.
- Quantum computers could theoretically halve AES key strength using Grover's algorithm, making 256-bit keys equivalent to current 128-bit security.
- Developer-friendly libraries like Google Tink are recommended over hand-coding AES implementations to avoid common security mistakes.

[1] <https://cybergibbons.com/reverse-engineering-2/why-is-unauthenticated-encryption-insecure/>

[2] <https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html>

[3] <https://cloud.google.com/bigquery/docs/reference/standard-sql/aead-encryption-concepts>

[4] <https://qz.com/674520/companies-are-sharing-their-secret-access-codes-on-github-and-they-may-not-even-know-it/>

6 Public Key Encryption and Digital Signatures: Unleashing RSA

This chapter covers

- Using RSA for encrypting data
- Using RSA for digital signatures

When you buy products online, look for directions to a restaurant, interact with friends and strangers on a social network, or collaborate with coworkers on a video conference call, you depend on public key cryptography. Without public key cryptography, the Internet as we know it would not exist.

Secure communication on the internet relies on the Transport Layer Security (TLS) protocol, which is built upon public key cryptography algorithms. Mastering TLS is a vital skill for any developer. With a solid understanding of public key cryptography, configuring and troubleshooting TLS connections becomes straightforward. Without this knowledge, you may find yourself blindly copying commands from blogs, hoping your changes will fix issues. Public key cryptography is not just a tool—it's an essential foundation for any developer working with secure applications.

This chapter offers application developers a friendly introduction to public key cryptography. The goal is to teach you how to use public key cryptography to solve real-world security problems. The mathematics underlying public cryptography is extremely interesting but is beyond the scope of this book. There are no math equations in this chapter, just sample code that leverages widely used Java libraries to help you develop an intuitive understanding of public key cryptography and how to use it in your applications.

TIP

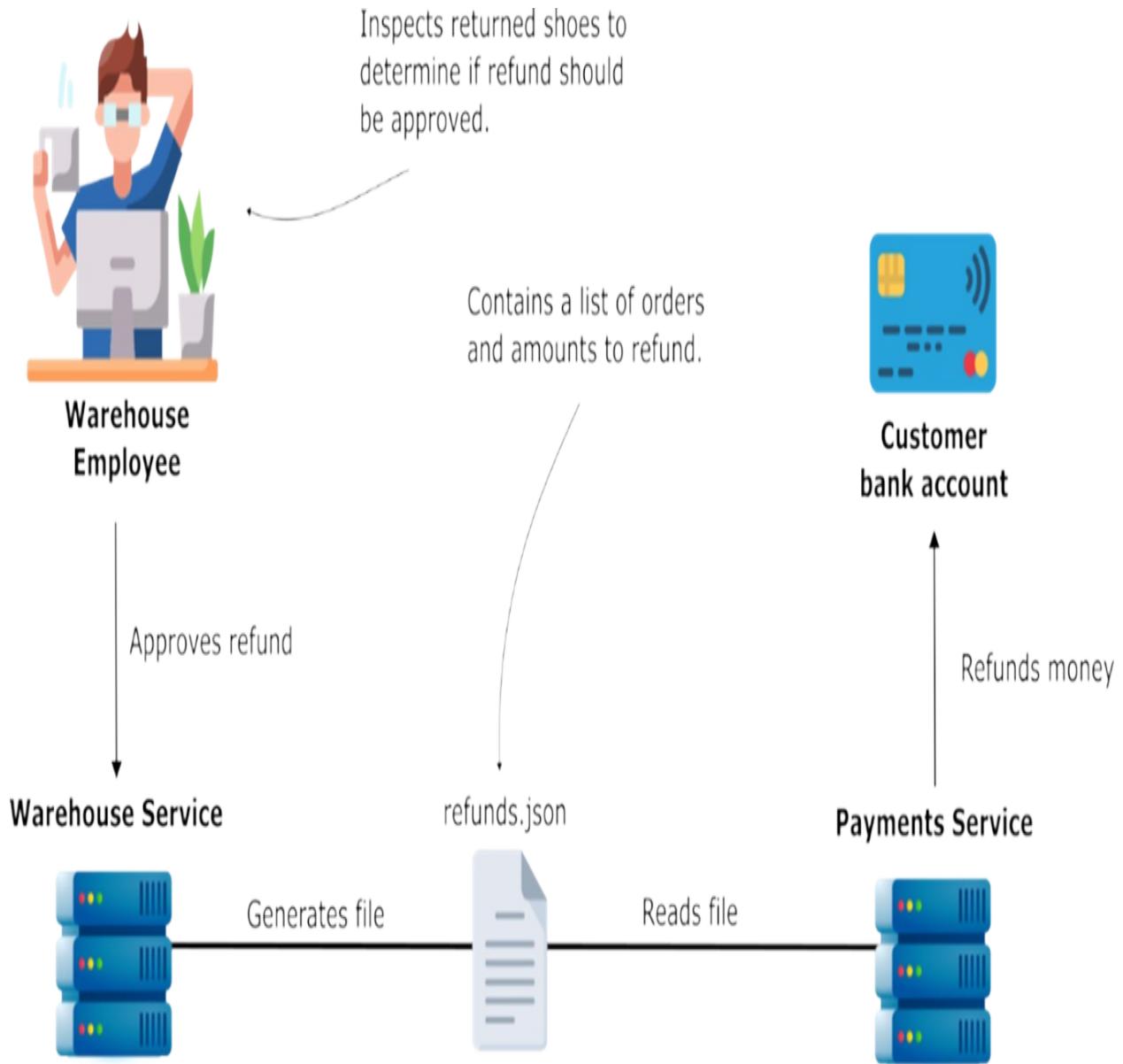
This chapter assumes you are familiar with content covered in chapter 3, 4,

and 5. In particular Hashed Message Authentication Code (HMAC), Java Cryptography Architecture (JCA), Advanced Encryption Standard (AES), and authenticated encryption using Galois Counter Mode (GCM). If you need a refresher please review chapters 3,4 and 5.

6.1 The secret key distribution problem

To understand public key cryptography, we will start by exploring the key distribution problem using a variation of the ACME Inc. scenario we have been exploring in previous chapters. ACME Inc., an online shoe retailer, allows customers to return shoes they don't like for a full refund. Customers mail the returns to ACME Inc's warehouse where staff check that the returned items are in good condition and authorize a credit card refund using the warehouse management app (figure 6.1).

Figure 6.1 ACME Inc. staff inspect returned merchandise and approve refunds using the warehouse management service. The payment service issues refund to customer credit card accounts.



The warehouse management service creates a file with a list of all newly approved refunds. Once per day, the payments service gets this file from the warehouse to issue refunds to customer credit cards.

The next snippet shows an example of the `refunds.json` file's content.

```
[
  {
    "orderId" : "12345",
    "amount" : 500
  },
  {
    "orderId" : "6789",
  }
]
```

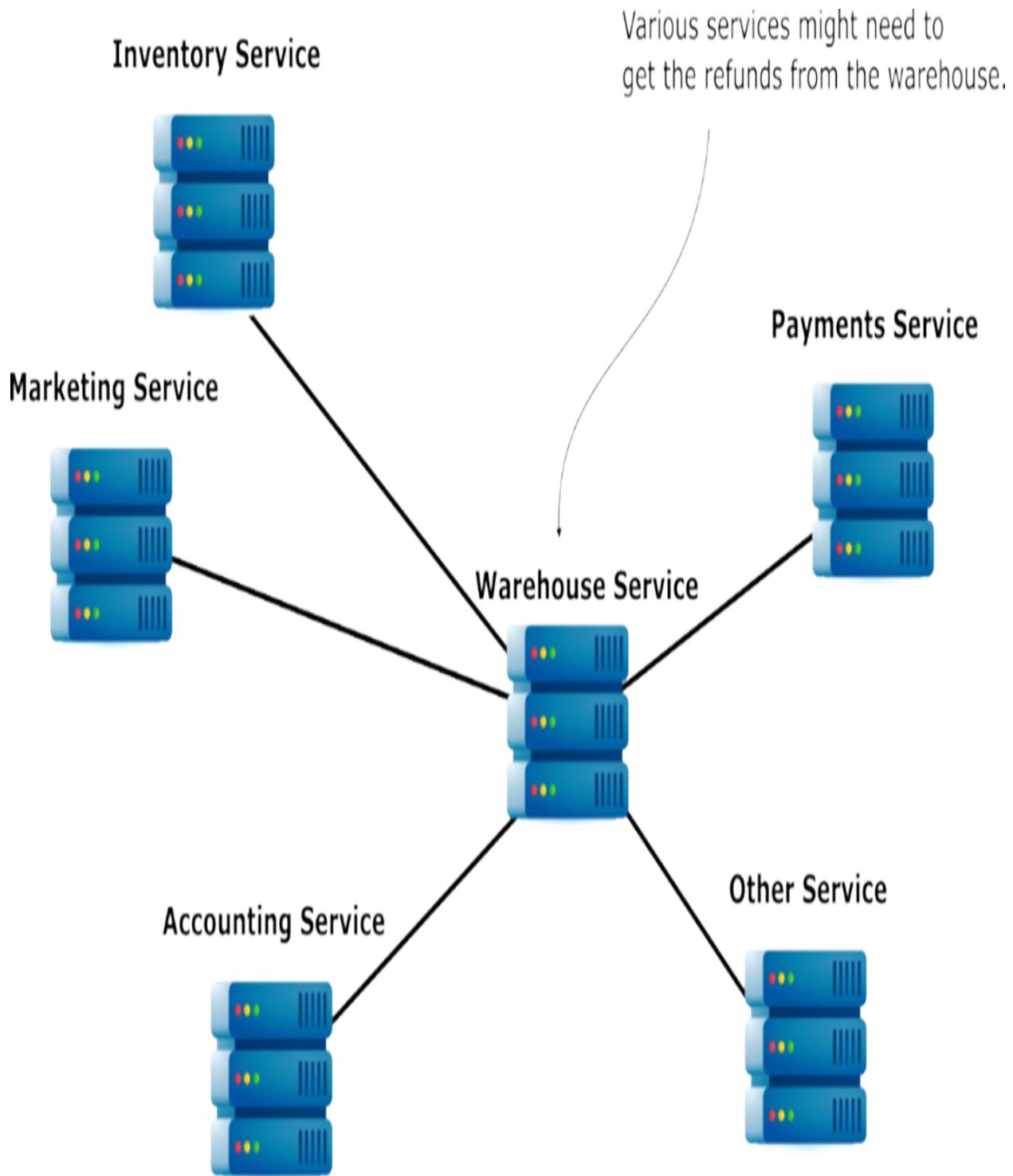
```
"amount" : 250  
} ]
```

In previous chapters we used a symmetric secret key which we stored in the Spring Boot application.properties configuration files of the warehouse and payment services.

Since the AES secret key is a configuration file an administrator must set the same key value in both the payments service application.properties and warehouse service application.properties file, this is not a hard thing to do if we only have two applications. However, multiple applications will need to talk to one another when the system grows. For example, the accounting application needs the data from the warehouse to keep the corporate books up to date.

The inventory management system needs access to the refunds to accurately track inventory. The marketing application needs access to the refunds to look for patterns of products that customers don't like (figure 6.2).

Figure 6.2 Usually in a service-oriented system, more than one application needs to talk to others. This fact complicates the distribution of the keys.



Administrators must set the secret key in all the applications that need to interact with the warehouse service. In an enterprise with hundreds of services managing encryption keys in configuration files is very hard, we need a better approach to key management.

Storing a secret key in plain text in the `application.properties` is very dangerous. A hacker can steal the configuration file and the encryption key, allowing the hacker access to encrypted data. Furthermore, once we know that a key has been stolen, we must assume that all keys have been stolen. Therefore, we must change all the encryption keys we use in all the configuration files across all the systems we are running without disrupting business operations; this is a huge effort. I know, I repeat myself, but it's important!

Note

Storing a secret key in plain text in the properties files or hardcoded is very dangerous. Never store secrets in configuration files. Instead, use proper vaults and infrastructure that properly protects secret values.

The problem of sharing a secret key between applications is called *the key distribution problem*. Solving the key distribution problem by storing keys in configuration files poses high management costs and security risks. We must find a way of sharing encryption keys without using configuration files. Public key cryptography provides the tools to solve the key distribution problem.

6.1.1 Exercises

1. What problem does public key cryptography solve compared to symmetric key cryptography?
2. Why is it safe to share a public key but not a private key?
3. If a hacker steals a private key, what should you do?
4. Why do systems use digital certificates when sharing public keys?

6.2 Public key cryptosystems

For centuries, governments, militaries, and diplomats have relied on encryption to protect their secrets. Early examples of encryption include the Spartans' use of the scytale, a simple device that wrapped a strip of parchment around a rod to encode messages. The Romans employed the Caesar cipher, shifting letters by a fixed number to disguise their

communications. These early methods, though primitive, laid the foundation for securing sensitive information.

As empires expanded and conflicts grew more complex, the challenge of key distribution emerged. Without secure ways to share encryption keys, the secrecy of messages could easily be compromised. In medieval Europe, royal courts and military commanders sent couriers carrying handwritten keys or codebooks to their counterparts. These couriers faced great peril, as enemy capture meant the entire cryptographic system could be exposed.

By the Renaissance, ciphers became more sophisticated. The Vigenère cipher, described as "unbreakable" for centuries, required both parties to possess the same keyword to encode and decode messages. Despite its ingenuity, it still relied on secure key exchange, often requiring trusted envoys or meetings to deliver these critical keys in person.

During the World Wars, encryption reached new heights of complexity. The Germans employed machines like the Enigma, relying on intricate key systems updated daily. Even here, the distribution of keys presented a monumental challenge. Codebooks were delivered under armed guard, often at great risk, and their loss or theft could jeopardize entire operations.

This long history of ingenuity and danger highlights the critical importance of secure key distribution. It wasn't until 1976, with Whitfield Diffie's and Martin Hellman's groundbreaking work, that the world saw a revolutionary shift. Their paper, *New Directions in Cryptography*, introduced the concept of *public key cryptography*, rendering the age-old reliance on secret courier systems obsolete. This innovation marked a turning point, paving the way for modern, secure communications in the digital era.

The key insight in public key cryptography is the introduction of *public* and *private* keys. Public keys are freely shared with the world and it is okay for anyone to possess a copy of the public key. Private keys are kept secret and never shared with anyone.

The private and public key form a pair that is mathematically bound to each other in such a way that the private key only works with its public key, and the public key only works with the private key. For example, plain text

encrypted with the public key can only be decrypted with the private key, and plain text encrypted with the private key can only be decrypted by the public key. Security rests on keeping the private key secret, if the private key is compromised then a new key pair must be generated. Figure 6.3 illustrates the concept of public-private key pairs.

Figure 6.3 A key pair consisting of a private and a public key. The keys are numbers that are related to each other using a mathematical equation. The private key only works with the public in the same key pair. The public key only works with the private key from the same key pair. The private key must be kept secret and the public key by shared with anyone including friends, enemies, and hackers. Security in a public key cryptosystem rest on keeping private keys private.

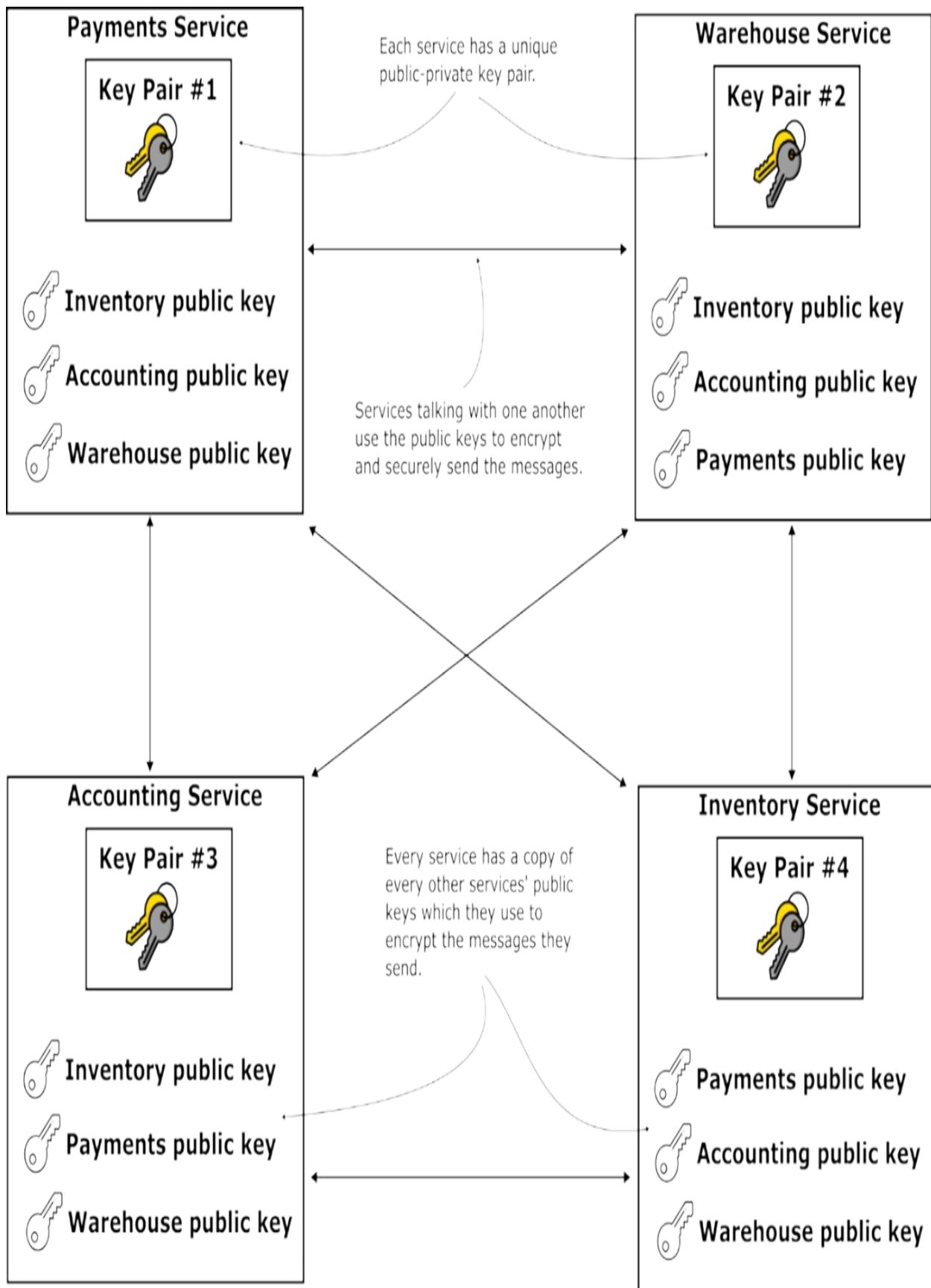
Key Pair



The public and private key are generated at the same time. They are mathematically related to each other and only work one with another.

In a system with many services, each service must have its own public-private key pair. The service's private key is stored securely in a key vault to protect it from theft. Two services wishing to communicate securely exchange public keys over an insecure network. Once the public keys are exchanged, they can be used to encrypt communications between the services (figure 6.4).

Figure 6.4 Each service has a unique public-private key pair, it freely shares its public key over the network to any other service that might ask for it. When two services want to communicate, they exchange public keys and use those keys to secure the communication link. If a hacker steals a private key, only the communication link using that private key is compromised. A hacker that only has access to the public keys cannot comprise the security of the system.



We assume that a hacker can read network traffic and modify it. For example, the hacker can intercept an HTTP request containing a public key, modify the value of the public key, and forward the HTTP request to the recipient. We encode the public key inside a digital certificate to protect against modification of a public key during transit. Digital certificates are covered in chapter 8, and they are essential for establishing trust that the public key being used is the correct one.

WARNING

Digital certificates are critical to the security of a public key cryptosystem. You must understand and get comfortable with digital certificates before using public key cryptography in a production application. To make the content in this chapter more straightforward to understand, we don't use digital certificates in the sample applications. Read the next chapter before using public key cryptography in your application.

Public key cryptosystems are built on top of a trap door mathematical function. A trap door function is one where it is fast to compute a result based on input parameters, but infeasible to compute the parameters given the result. For example, given two large prime numbers it is easy to multiple them quickly, however given the result of the multiplication it is infeasible to work out what the original prime numbers used in the multiplication. There are two widely used public key cryptography systems:

- RSA - based on the difficulty of factoring the product of two large prime numbers (we discuss RSA in this chapter).
- *Elliptic Curves* - based on the challenge of solving the elliptic curve discrete logarithm problem (which we discuss in chapter 7).

The rest of this chapter provides sample applications using RSA to encrypt and sign content using plain Java. We will cover Elliptic Curves in the next chapter as well. The samples focus on exposing important usage patterns of public key cryptography.

6.2.1 Exercises

5. What is the math problem that RSA security is based on?
6. Why would quantum computers break RSA?
7. Which RSA key sizes are considered safe today?
8. What is the difference between RSAES-PKCS1-v1_5 and RSA-OAEP?

6.3 RSA public key cryptosystem

The RSA public key encryption algorithm was invented in 1977, it is named after the initials of its inventors Ron Rivest, Adi Shamir, and Leonard Adleman. RSA also refers to a company called RSA Security LLC that was founded in 1982 by the inventors of the RSA algorithm to commercialize their invention. Being first to market with security products based on the RSA algorithm led to widespread adoption and implementation of the RSA cryptosystem in all programming languages and numerous security protocols.

The mathematical details of RSA are beyond the scope of this book. As a developer it is important to know the following facts about the RSA algorithm.

- The RSA algorithm is based on the mathematical problem of factoring a large integer into a product of prime numbers.
- Integer factoring algorithms are very slow when the number being factored is very large. The slow speed of integer factoring is the basis of security for the RSA algorithm.
- If mathematicians discover a fast integer factoring algorithm, then RSA will no longer be secure.
- Computers are getting faster, and factoring algorithms implementations are improving. At the time of writing the largest number ever factored is a 250-decimal digit (829 bit) number from the RSA factoring challenge called RSA-250.
- Quantum computers will break RSA because they can factor integers much more quickly than classical computers. However, at the time of writing, quantum computers are not powerful or widespread enough to be a practical threat.

6.3.1 Configuring RSA

Now that we've covered the theory, let's dive into configuring RSA. When using RSA, there are two critical settings you must choose:

- Key Size: Determines the length of the key, typically measured in bits (e.g., 2048 or 4096). Larger keys offer stronger security but require more computational resources.
- Padding Scheme: Protects against certain cryptographic attacks by adding random data to the plaintext before encryption. Common schemes include PKCS#1 v1.5 and OAEP (Optimal Asymmetric Encryption Padding).

The larger the key size, the more security you get. Today, RSA 2048-bit keys are still considered secure, but you should use keys that are longer, 3072-bit or 4096-bit. Your corporate security standards should provide guidance on the minimum key length for RSA.

There are two RSA-based encryption schemes:

- RSAES-PKCS1-v1_5 - An older RSA encryption scheme that uses PKCS #1 v1.5 padding. It's simple but considered less secure against chosen-ciphertext attacks, so it's mostly replaced by OAEP in modern systems.
- Optimal Asymmetric Encryption Padding (OAEP) - A padding scheme for RSA that mixes the message with random data using hash functions. It provides stronger security guarantees and is recommended instead of PKCS1 v1.5.

Using RSAES-PKCS1-v1_5

Besides having a funny name, RSAES-PKCS1-v1_5 is an RSA encryption scheme. It was defined in the Public Key Cryptography Standards (PKCS) version 1.5 and later in RFC 2313. While it was widely adopted at the time, researchers eventually discovered serious vulnerabilities that made it insecure. As a result, you should never use it for encryption today.

Unfortunately, many older systems and products still support RSA PKCS#1 v1.5 for the sake of backward compatibility. This creates a risk of downgrade

attacks, where an attacker tricks a system into using this older, less secure protocol instead of a more secure one. By exploiting these weaknesses, attackers can compromise the confidentiality of encrypted messages.

To protect your systems and data, configure your applications to reject PKCS#1 v1.5 completely. This ensures you are not exposed to downgrade attacks and rely only on secure encryption schemes.

Using Optimal Asymmetric Encryption Padding (OAEP)

Optimal Asymmetric Encryption Padding (OAEP), was specifically designed to fix the security weaknesses in PKCS#1 v1.5. One of the key improvements OAEP provides is the addition of randomized padding to the plaintext before encryption. Padding involves adding extra data to the original message to make it a fixed length, but in OAEP, this padding is randomized.

This randomness ensures that even if the same plaintext is encrypted multiple times, the resulting ciphertext will be different each time. This is important because predictable patterns in plaintext—such as repeated structures or identical encrypted messages—can give attackers clues to reverse-engineer the encryption (remember the AES with CBC mode we discussed in chapter 5). As an interesting note, this is actually how Americans (through Alan Turing's research) broke the German Enigma machine messages during the Second World War.

OAEP eliminates these patterns by randomizing the padding, making it far more difficult for an attacker to analyze and exploit the ciphertext.

These improvements significantly strengthen RSA encryption by preventing vulnerabilities like chosen ciphertext attacks, where an attacker manipulates ciphertexts to extract information about the plaintext. With its robust design, OAEP has become the standard padding scheme for RSA encryption, ensuring stronger protection for sensitive data.

If you are developing or maintaining applications, you should always use RSA-OAEP as the padding scheme for RSA encryption. It is widely regarded as the modern, secure standard for ensuring the confidentiality of your data

and protecting against potential exploits.

TIP

Always use RSA-OAEP. Turn off RSA-PCKSv1.5 from anything you are using. Use RSA 4096-bit keys for maximum security. Many professional cryptographers prefer Elliptic Curve Cryptography (ECC) over RSA make sure to read also chapter 7 on ECC.

6.3.2 Hybrid Encryption

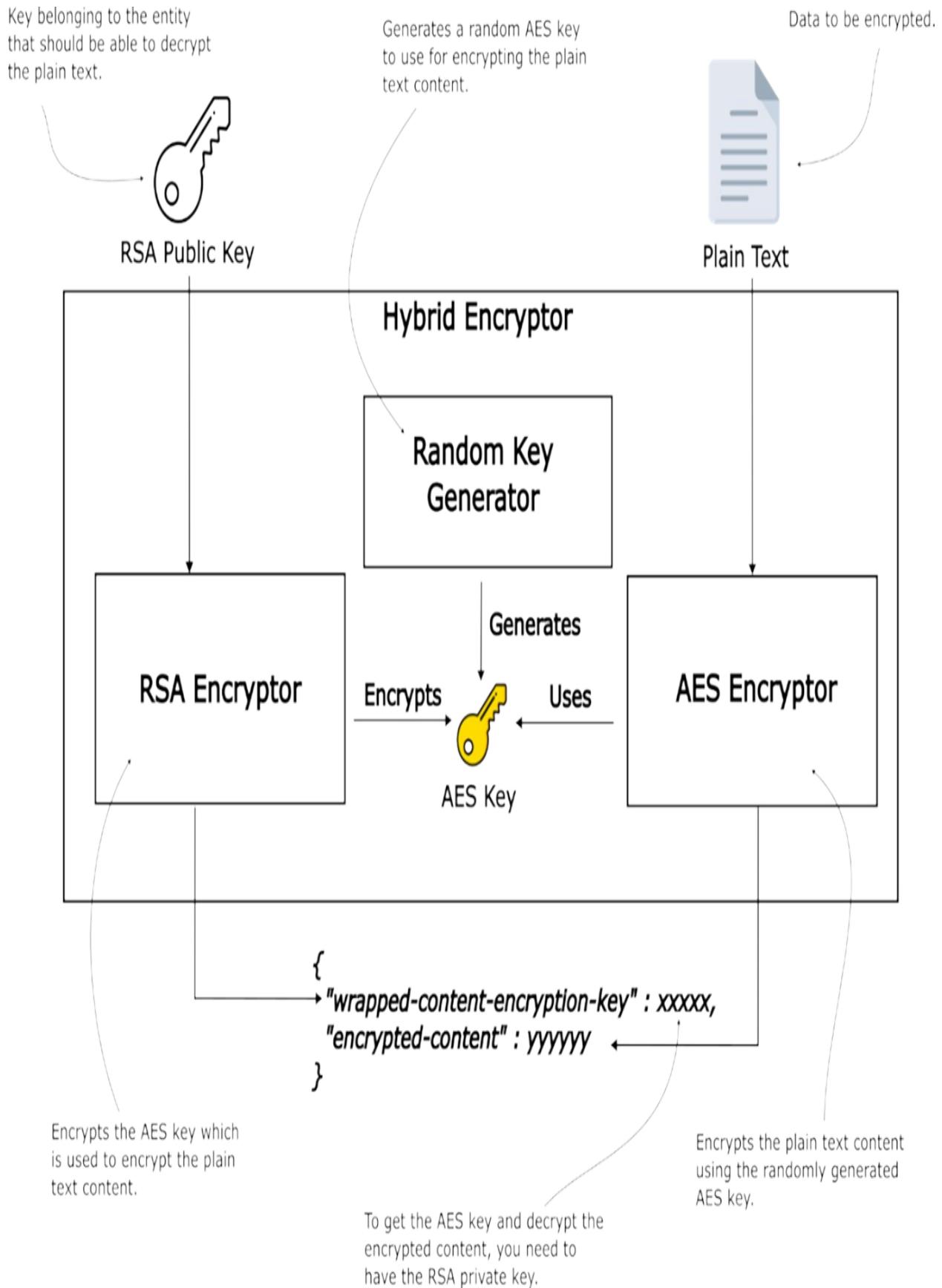
The implementation of the RSA encryption algorithm is significantly slower than that of AES (which we discussed in chapter 5). Especially because processors provide special instructions for hardware-accelerated AES, and the mathematics for RSA encryption is CPU intensive. Therefore, combining AES with RSA into a hybrid symmetric and asymmetric scheme is common.

The key idea is to use AES for content encryption and then RSA to encrypt the AES encryption key. Key wrapping is the terminology used to indicate that an encryption key is itself encrypted. The hybrid encryption and decryption consists of the steps below (figure 6.5).

- Sender encryption process
 - a. Generate a random AES key called the Content Encryption Key (CEK)
 - b. Execute AES encryption on content using the CEK from step 1 as the key
 - c. Wrap the CEK by encrypting it with the RSA public key
 - d. Send the encrypted CEK along with the encrypted content to the recipient
- Recipient decryption processes
 - a. Unwrap the CEK by decrypting it with the RSA private key
 - b. Execute AES decryption on content using the CEK to recover original content

Figure 6.5 Hybrid encryption using a high-performance symmetric algorithm like AES to encrypt content. The AES Content Encryption Key (CEK) is then encrypted with the recipient's RSA public key, allowing the recipient to use their private key to decrypt the CEK and then decrypt

the encrypted content.



Widely deployed security protocols such as TLS, IPsec, JWE, SSH use hybrid encryption. Chapter 10 covers TLS and chapter 11 covers JSON Web Encryption (JWE) with RSA which implements the hybrid encryption scheme we just described. When using hybrid encryption, you will have to configure two algorithms:

- AES for bulk data content encryption. Review chapter 5 on AES for best practices for configuring AES. AES-GCM with 256-bit key is an example configuration.
- RSA for wrapping the AES Content Encryption Key (CEK). Always use RSA-OAEP mode. Key sizes should be 2048-bit or higher.

6.3.3 Signing data with RSA

In chapter 4 we discussed Message Authentication Codes (MACs) and Hash-based Message Authentication Codes (HMACs). They are great tools for checking integrity and authentication, and they work with a shared secret key (symmetric key).

Digital signatures, on the other hand, also provide integrity and authentication, but they are built on asymmetric key pairs (a private key and a public key). Just like with encryption, this approach has the advantage of making key distribution easier and helping to apply the principle of least privilege more effectively.

Important

The private key is always used for the sensitive action. In the case of encryption and decryption, the private key is used for decryption. Anyone can encrypt data with the public key, but only the holder of the private key can decrypt it. When it comes to digital signatures, the private key is used for signing. The matching public key is then used for verification. This way, anyone can check the signature using the public key, but only the owner of the private key can create the signature.

Figure 6.6 Using RSA for digital signatures. The Warehouse Service signs data with its private key, and the Payments Service (or any other service) can verify the signature using the

corresponding public key. This ensures integrity and authenticity, since only the service holding the private key can create the signature.



Data

A service can sign data using the private key of the key pair. Only the service who owns the data to be signed should have the private key.

Warehouse Service



RSA Private Key

Any other service can check the signature using the public key of the key pair.



Data



RSA Public Key

Payments Service



One important difference is non-repudiation. Non-repudiation means that once someone has signed a message, they cannot later deny that they created it. Digital signatures give you this property, because only the owner of the private key could have produced the signature, while anyone can verify it using the public key. With HMACs, this is not possible, since both sides share the same secret key — either of them could have created the code, so you cannot prove who actually did it.

Another key difference is performance. HMACs are very fast because they are based on hashing, which is lightweight and efficient even for large amounts of data. Digital signatures, in contrast, are slower since they use heavy math from public key cryptography. This makes HMACs a good choice when speed matters, such as for signing many API requests, while digital signatures are better suited for cases where performance is less critical but strong guarantees (like non-repudiation) are needed.

6.3.4 Exercises

9. Why is RSA often combined with AES in a hybrid scheme?
10. What is the term for encrypting an encryption key itself?
11. List the steps a sender follows in hybrid encryption.
12. Which widely used protocols rely on hybrid encryption?
13. Which key of the key pair (private or public) is used for signing data?
14. List the main important differences between HMAC and RSA digital signatures.

6.4 Java Support for RSA

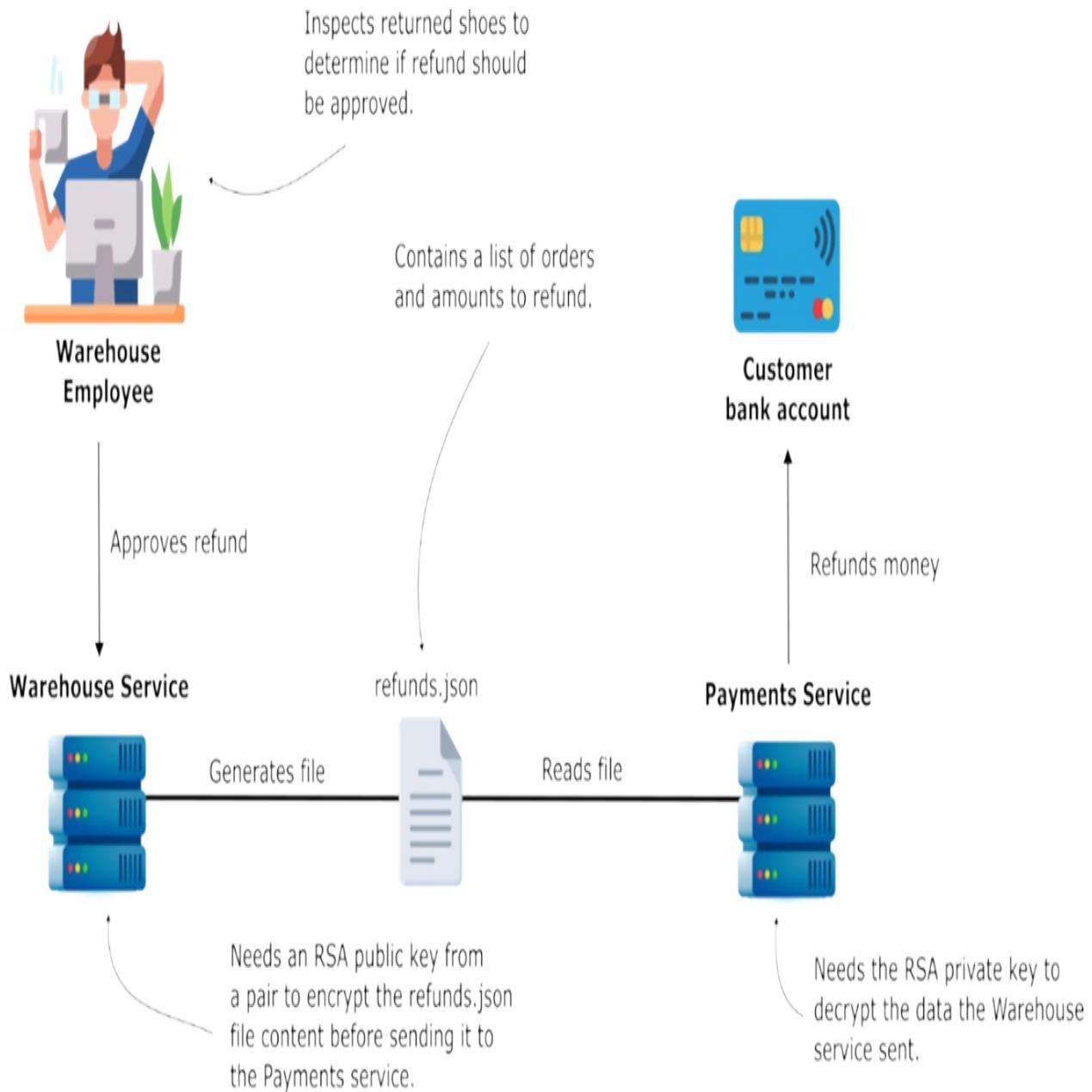
Now let's get back to our Acme Inc. scenario. Using AES to encrypt the `refunds.json` file, as we did in chapter 5, works fine. But Acme Inc. engineers now face two problems:

- how to handle key rotations
- the risk of keeping the same secret key on both sides of the communication channel

“This gives too much control to the Payments service and goes against the Zero-Trust principle,” they say.

So, they decided to redesign the code and use RSA instead. With RSA, key rotations are easier because they can simply create an endpoint that exposes the public keys, instead of manually updating each service. This approach also avoids the need to share secret keys everywhere.

Figure 6.7 The Warehouse service uses the public key from a key pair to encrypt the content of the refunds.json file. The Payments service uses the private key from the pair to decrypt the data it receives.



In a real-world setup, the Payments service would hold the key pair (both the private and the public key). It would then expose the public key through an endpoint. This way, you don't need to rotate keys in two places (and we'll discuss more about this in chapters 11 and 12).

But to keep this example simple (for now), I decided to generate a key pair separately and then configure:

- the public key in the Warehouse service

- the private key in the Payments service

This example will teach you:

- how to generate an RSA key pair with plain Java
- how to encrypt data with RSA using a public key in plain Java
- how to decrypt data with RSA using a private key in plain Java

Here are the steps we'll follow:

1. Generate an RSA key pair using a small Java application we call the key generator.
2. Encrypt the refunds details in the Warehouse application using the public key from step 1.
3. Decrypt the refunds details in the Payments application using the private key from step 1.

Listing 6.1 shows how to generate a key pair with Java. You can find the full implementation in the project ssfd_ch6_ex1-keygenerator. Running this simple app will save the public and private keys of a pair in the root folder of the project. Then, you configure the public key in the Warehouse service (ssfd_ch6_ex1-warehouse) and the private key in the Payments service (ssfd_ch6_ex1-payments).

Listing 6.1 Generating an RSA key pair

```
public class RsaKeyPairGenerator {

    private final int keySize;
    private final SecureRandom secureRandom;

    public RsaKeyPairGenerator(int keySize, SecureRandom secureRand
        this.keySize = keySize;
        this.secureRandom = secureRandom;
    }

    public KeyPair generate() throws Exception {
        KeyPairGenerator kpg =
            KeyPairGenerator.getInstance("RSA");  #A
        kpg.initialize(keySize, secureRandom);  #B
        return kpg.generateKeyPair();  #C
    }
}
```

```
    }  
}
```

The key generator application saves the keys in two separate files. Take the public key file and place it in the resources folder of the Warehouse application.

The Warehouse application also includes an example refunds.json file, similar to the ones we used in earlier chapters. When you run the Warehouse application (a simple Java app), it will create a new file in the root folder of the project. This file, called refunds.json.rsa, stores the RSA-encrypted content.

The next snippet shows an example of how a refund looks in plain text:

```
[  
  {  
    "orderId": "10001",  
    "amount": 120  
  }  
]
```

Listing 6.2 show the Java code doing the RSA encryption. You can find the full app implementation in the ssfd_ch6_ex1-warehouse project provided with the book.

Listing 6.2 Encrypting using the RSA public key

```
public static byte[] rsaEncryptChunked(  
    byte[] data,  
    PublicKey key) throws Exception {  
  
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding"); #A  
    cipher.init(Cipher.ENCRYPT_MODE, key); #B  
  
    int modulusBytes = ((RSAPublicKey) key).getModulus().bitLength()  
    modulusBytes = (modulusBytes + 7) / 8; #C  
    int maxBlock = modulusBytes - 11; #C  
  
    ByteArrayOutputStream out = new ByteArrayOutputStream(); #D  
    for (int off = 0; off < data.length; off += maxBlock) { #D  
        int len = Math.min(maxBlock, data.length - off); #D
```

```

        out.write(cipher.doFinal(data, off, len)); #D
    }

    return out.toByteArray(); #E
}

```

Now take the encrypted content file and copy it into the resources folder of the Payments service. Also place the private key file you generated earlier in the same folder.

When you run the Payments application, it will use the private key to decrypt the encrypted content and produce the plain text file.

Listing 6.3 Decrypting using the RSA private key

```

public static byte[] rsaDecryptChunked(
    byte[] ciphertext,
    PrivateKey privateKey) throws Exception {

    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding"); #A
    cipher.init(Cipher.DECRYPT_MODE, privateKey); #B

    int keySizeBytes = #C
        (((RSAPrivateKey) privateKey).getModulus().bitLength() + 7) /

    try (ByteArrayOutputStream baos = new ByteArrayOutputStream())
        for (int offset = 0; offset < ciphertext.length;) { #D
            int len = Math.min(keySizeBytes, ciphertext.length - offset);
            baos.write(cipher.doFinal(ciphertext, offset, len)); #D
            offset += len;
        }
        return baos.toByteArray(); #E
    }
}

```

This example shows a straightforward way to use RSA encryption with plain Java. Most programming languages also provide standard cryptographic algorithms, so you can apply the same ideas to implement RSA encryption and decryption in other technologies as well.

Let's now extend our scenario. For this demo, we want the Warehouse app to sign the encrypted content using RSA. The Payments app will then first

verify the signature before it tries to decrypt the data.

I have separated this example in projects ssfd_ch6_ex2-warehouse and ssfd_ch6_ex2-payments provided with this book. The steps you need to follow are:

1. Generate a new key pair using the key pair generator app (project ssfd_ch6_ex1-keygenerator).
2. Copy the private key in the Warehouse application and name it signing_private_key.pem
3. Copy the public key in the Payments application and name it signing_public_key.pem

Remember: encryption uses the public key, while signing uses the private key. This means the Warehouse app will use a public key to encrypt data but a private key to sign it.

Pay close attention: the apps actually use two separate key pairs — one pair for encryption/decryption and another pair for signing/verification. It would not make sense to use the same key pair for both tasks. If we did, then both apps would need access to both the private and public keys, which would reduce security and make the setup almost the same as using a single shared secret. In fact, it would be even worse than a symmetric approach, because asymmetric algorithms like RSA are slower and less efficient.

Listing 6.4 shows you how the Warehouse app signs the refunds data using a plain Java implementation.

Listing 6.4 Signing the data with RSA using a private key

```
public static byte[] signSha256Rsa(
    byte[] data,
    PrivateKey privateKey) throws Exception {

    Signature sig = Signature.getInstance("SHA256withRSA");  #A
    sig.initSign(privateKey);  #B
    sig.update(data);  #C

    return sig.sign();  #D
}
```

Listing 6.5 shows you how to use plain Java to verify an RSA signature using the public key from the pair. This is what the Payments service do before decrypting the data. If the signature cannot be verified, the service rejects the data.

Listing 6.5 Verifying the RSA signature using the public key

```
public static boolean verify(
    byte[] data,
    byte[] signature,
    PublicKey publicKey) throws Exception {

    Signature sig = Signature.getInstance("SHA256withRSA"); #A
    sig.initVerify(publicKey); #B
    sig.update(data); #C

    return sig.verify(signature); #D
}
```

That's all for asymmetric key encryption and signing with RSA for now. You've seen how RSA can be used both for encrypting data and for creating digital signatures, as well as its strengths and limitations. In the next chapter, we'll continue with another form of asymmetric encryption: Elliptic Curve Cryptography (ECC). ECC uses a different mathematical foundation but aims to provide the same guarantees with smaller keys and better performance.

6.5 Answers to exercises

1. What problem does public key cryptography solve compared to symmetric key cryptography?
It solves the key distribution problem — you no longer need to secretly share the same key with every service.
2. Why is it safe to share a public key but not a private key?
Because the public key only encrypts or verifies, it cannot unlock the data by itself. The private key must stay secret to protect the system.
3. If a hacker steals a private key, what should you do?
Generate a new key pair and replace the compromised one.
4. Why do systems use digital certificates when sharing public keys?
Certificates protect against tampering and impersonation by proving the

public key really belongs to the claimed service.

5. What is the math problem that RSA security is based on?

It is based on the difficulty of factoring large numbers into their prime factors.

6. Why would quantum computers break RSA?

Quantum computers can run algorithms that factor numbers much faster than classical computers.

7. Which RSA key sizes are considered safe today?

2048-bit keys are safe today, but security experts recommend 3072 or 4096-bit for long-term use.

8. What is the difference between RSAES-PKCS1-v1_5 and RSA-OAEP?

PKCS1 v1.5 is older and weaker, vulnerable to chosen-ciphertext attacks. OAEP uses random padding, making it the modern and secure option.

9. Why is RSA often combined with AES in a hybrid scheme?

Because RSA is slow for large data, while AES is fast and efficient.

10. What is the term for encrypting an encryption key itself?

This is called key wrapping.

11. List the steps a sender follows in hybrid encryption.

Steps:

- a. Generate an AES key.
- b. Encrypt the data with AES.
- c. Encrypt the AES key with RSA
- d. Send both the encrypted data and the encrypted key

12. Which widely used protocols rely on hybrid encryption?

Protocols like TLS, IPsec, JWE, SSH all use hybrid encryption.

13. Which key of the key pair (private or public) is used for signing data?

The private key is used to sign the data. Only the owner of the data has the private key and this way it ensures non-repudiation.

14. List the main important differences between HMAC and RSA digital signatures.

Keys

HMAC: uses a shared secret key (symmetric).

RSA: uses a public/private key pair (asymmetric).

Who can verify

HMAC: only parties that share the secret key.

RSA: anyone with the public key.

Non-repudiation

HMAC: not possible (both sides share the same key).

RSA: possible (only the private key holder could sign).

Performance

HMAC: very fast, based on hashing.

RSA: slower, uses heavy math.

Use cases

HMAC: efficient checks between trusted parties (e.g., API calls).

RSA: signatures that must be verified by many, or require proof of authorship (e.g., certificates, software signing).

6.6 Summary

- Public key cryptography solves the key distribution problem by using mathematically related public-private key pairs where public keys can be freely shared while private keys remain secret.
- RSA public key encryption is based on the mathematical difficulty of factoring large numbers into prime factors, with security depending on this computational challenge.
- RSA keys should be at least 2048 bits today, with 3072 or 4096 bits recommended for long-term security against advancing computational power.
- RSAES-PKCS1-v1_5 is an older, insecure RSA padding scheme that should be avoided, while RSA-OAEP provides secure randomized padding and is the recommended standard.
- Hybrid encryption combines fast AES for content encryption with RSA for encrypting the AES key, providing both performance and security benefits.
- Key wrapping refers to encrypting an encryption key itself, commonly used in hybrid schemes where RSA encrypts AES content encryption keys.
- RSA digital signatures use the private key to sign data and the public key to verify signatures, providing integrity, authenticity, and non-repudiation.
- Digital signatures differ from HMACs by enabling non-repudiation since only the private key holder can create valid signatures, while anyone can verify with the public key.

- RSA signatures are slower than HMACs due to heavy mathematical operations but provide stronger guarantees including proof of authorship for legal purposes.
- Separate RSA key pairs should be used for encryption/decryption versus signing/verification to maintain security and avoid compromising either function.
- Private keys are always used for sensitive operations like decryption and signing, while public keys are used for encryption and signature verification.
- RSA encryption requires splitting large data into chunks since RSA can only encrypt data smaller than the key size minus padding overhead.
- Quantum computers could break RSA by efficiently factoring large numbers, making current key sizes inadequate against future quantum threats.
- Digital certificates are essential for validating public key authenticity and preventing man-in-the-middle attacks during key exchange.
- Sharing symmetric encryption and signing keys between different applications possess very high management costs and security risks.
- Public key cryptography algorithms use key pairs to perform encryption and signing operations.
- Key pairs have two mathematically related keys called the private and public key. Public keys are freely shared with the world it is okay for anyone to possess a copy of the public key. Private keys are kept secret and never shared with anyone.
- RSA and Ecliptic Curve Cryptography (ECC) are the two mostly widely used public key cryptosystems.
- Quantum computers will break RSA and ECC because they solve the mathematical problems RSA and ECC use much more quickly than classical computers. However, quantum computers are not powerful or widespread enough to be a practical threat.
- Key warping describes a widely used technique where a content encryption key is used to encrypt some content. Then the content encryption key is itself encrypted to protect the content encryption key during transit or storage.
- A hybrid encryption scheme combines symmetric and public key cryptography. A symmetric cipher such as AES is used to encrypt content. Then, the AES content-encryption key is wrapped by

encryption using RSA.

- RSA signatures use the private key to sign and the public key to verify. This ensures that only the private key holder can create a valid signature, while anyone can check it with the public key.
- Digital signatures provide integrity, authenticity, and non-repudiation. With HMAC, you get integrity and authentication, but not non-repudiation since both sides share the same secret.
- RSA signing and RSA encryption use different key pairs. One pair should be used for encryption/decryption and another for signing/verification to avoid weakening security.
- RSA signatures are slower than HMACs. They rely on heavier math, so they are often reserved for cases where non-repudiation or public verification is required.
- Always consult with your Information Security team to ensure you use corporate recommended configurations of common cryptographic algorithms.

7 Public Key Encryption and Digital Signatures: Using ECC

This chapter covers

- Using elliptic curve encryption
- Using elliptic curve digital signature
- Selecting a public key cryptosystem: RSA vs. elliptic curve

In chapter 6, we explored the mechanics of RSA, a foundational cryptographic system that has safeguarded digital communications for decades. But while RSA is the wise elder of public key cryptography, it has its challenges—particularly when it comes to performance. In this chapter, we shift gears to focus on Elliptic Curve Cryptography (ECC), a sleek, modern alternative that offers the same level of security with far smaller keys and better performance.

Before diving in, let's take a moment to appreciate how far we've come. Back in the 1970s, Whitfield Diffie and Martin Hellman introduced the world to public key cryptography, sparking a revolution. Little did they know that decades later, we'd be exploring mathematical curves to protect everything from online shopping to encrypted cat memes. As cryptographers like to joke, ECC is like trading in your vintage station wagon (RSA) for a high-performance sports car—sleek, efficient, and built for the demands of the modern world.

So, grab your gear! We'll first unravel the mysteries of ECC, explore how it builds on concepts from RSA, and learn why it has become the cryptographic tool of choice in the age of smartphones, cloud computing, and high-stakes digital security.

7.1 Elliptic curve public key cryptosystems

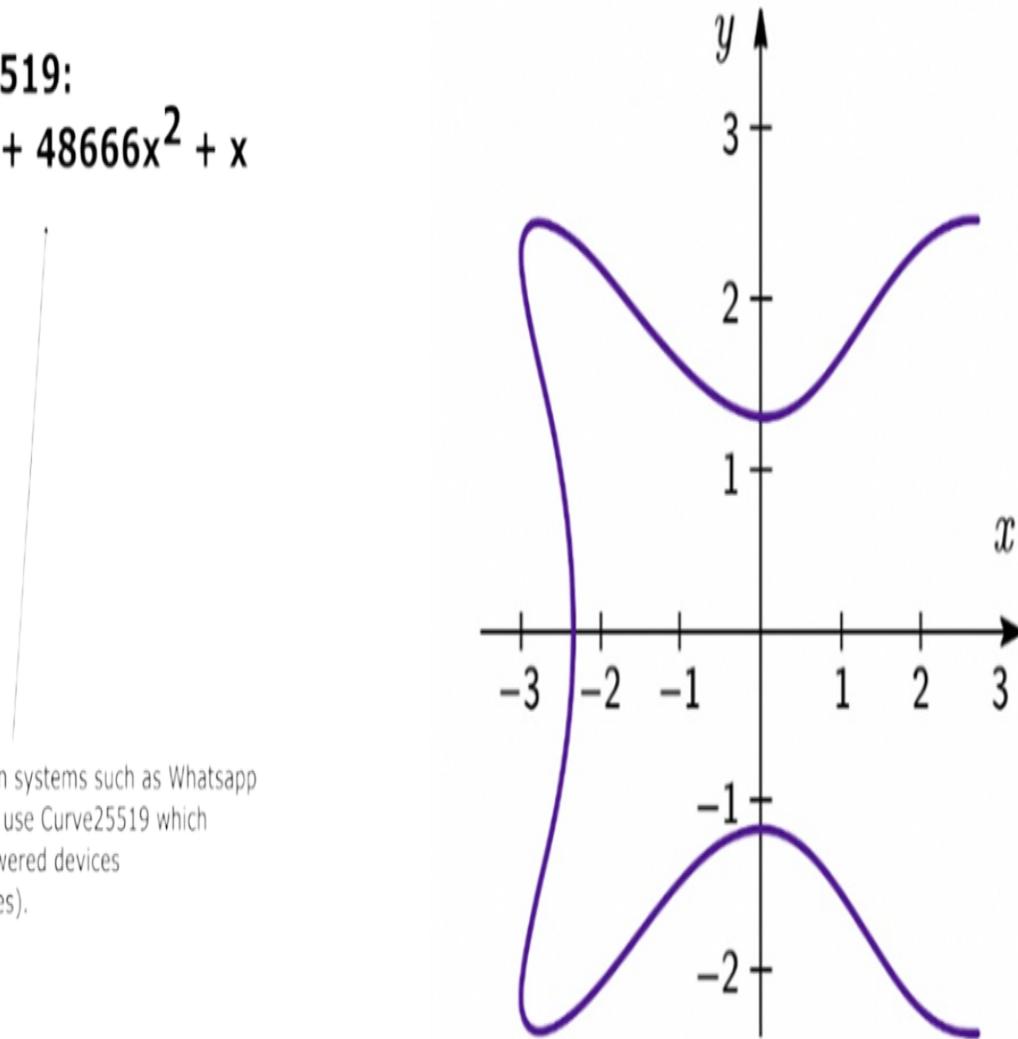
Elliptic Curve Cryptography (ECC) was invented in 1985 by Victor Miller and Neal Koblitz. It started gaining widespread adoption in 2004 as an alternative to RSA because it provides better performance. The mathematics of elliptic curve cryptography is beyond the scope of this book; otherwise, we'd shortly find smoke getting out through our ears. As a developer, knowing the following facts about the mathematics behind the ECC algorithms is important.

ECC is based on the mathematical problem of computing the discrete logarithm of a point on an elliptic curve (figure 7.1). If you are not a mathematician, don't bother with the equations behind and fancy math notions. But I leave this here since this gives the tool the difficulty of solving this problem, which is what makes ECC secure. An attacker would need immense computational power to figure out the solution.

- Computing the discrete logarithm of a point on an elliptic curve is very slow. The slow speed of computing the discrete logarithm of a point on an elliptic curve is the basis for the security of ECC.
- If mathematicians discover a fast way to solve the discrete logarithm problem, then ECC will no longer be secure.
- Quantum computers will break elliptic curve cryptography. However, yet, quantum computers are not powerful or widespread enough to be an immediate threat.

Figure 7.1 An example of an elliptic curve. This curve named **Curve25519** is one of the most used with ECC algorithm.

Curve25519:
 $y^2 = x^3 + 48666x^2 + x$



Secure communication systems such as WhatsApp and Signal commonly use Curve25519 which is efficient for low-powered devices (such as smart phones).

7.1.1 Configuring ECC

Imagine an elliptic curve as a fancy type of graph, drawn by solving a mathematical equation. Elliptic curves are special because they have unique mathematical properties that allow operations like "adding" two points or "multiplying" a point by a number, all while staying on the curve. These operations form the backbone of elliptic curve cryptography (ECC).

There are countless ways to define an elliptic curve by tweaking the variables of the equations. Each tweak gives you a new curve. However, not all curves are safe for cryptography:

- Some curves can be broken more easily, meaning attackers could potentially figure out your secret keys.
- To ensure security, cryptographers carefully test and approve specific curves for use in encryption.

Note

Not all mathematically possible elliptical curves can be used with cryptography. Cryptographers carefully test and approve specific curves for use in encryption.

To make life easier for developers, these approved curves are given names (sometimes long and strange rather than friendly) instead of long equations. For example:

- Curve25519: Known for its speed and security, commonly used in secure messaging apps.
- Curve448-Goldilocks: Offers very high security.
- P-256: A widely used curve standardized by the U.S. government.
- secp256k1: Famous for being used in Bitcoin and other cryptocurrencies.

For ECC to work, both the sender and receiver must use the same elliptic curve. If they don't, their encrypted messages won't make sense to each other —like trying to play a piano duet when one person is using the wrong sheet music.

When setting up encryption, you configure which elliptic curve to use. This decision boils down to choosing from the pre-approved, named curves, based on your needs. Cryptographers and security experts consider several trade-offs when selecting a curve:

- *Security level*, different curves have different security levels. For example, Curve25519 provides the equivalent of 128-bits of security, while Curve448-Goldilocks provides 224-bits of security.
- *Computational efficiency*, some curves have special mathematical properties that make them computationally efficient. Therefore, choice of curve impacts performance. For example, Curve25519 is faster and

more efficient than the P-256 curve.

- *Implementation safety*, some curves are harder than others to code increasing the risk of implementation bugs that affect security.
- *Trust in the curve designer*, there is an infinite number of elliptic curves but only a few that are standardized. When using a specific curve, you are trusting that the designer of the curve to pick a secure curve. It is best to use a curve that is widely peer reviewed and considered safe by many independent cryptographers.

The choice of curve has real-world implications for security. Now, Curve25519, designed by Daniel J. Bernstein, an independent cryptographer, is widely used and considered to have good security and performance. Consult your corporate security standards to see which curves are recommended for use in your application.

7.1.2 Diffie-Helman key agreement

Elliptic curve encryption combines two techniques to keep your data secure. As discussed in chapter 6, This is called a hybrid encryption scheme because it uses two different methods:

- AES (Advanced Encryption Standard): This is a fast and secure way to encrypt the actual content, like your message or file.
- Elliptic Curve Diffie-Hellman (ECDH): This is a clever way to generate the AES encryption key securely so that only the intended parties can use it, even if someone is watching the communication.

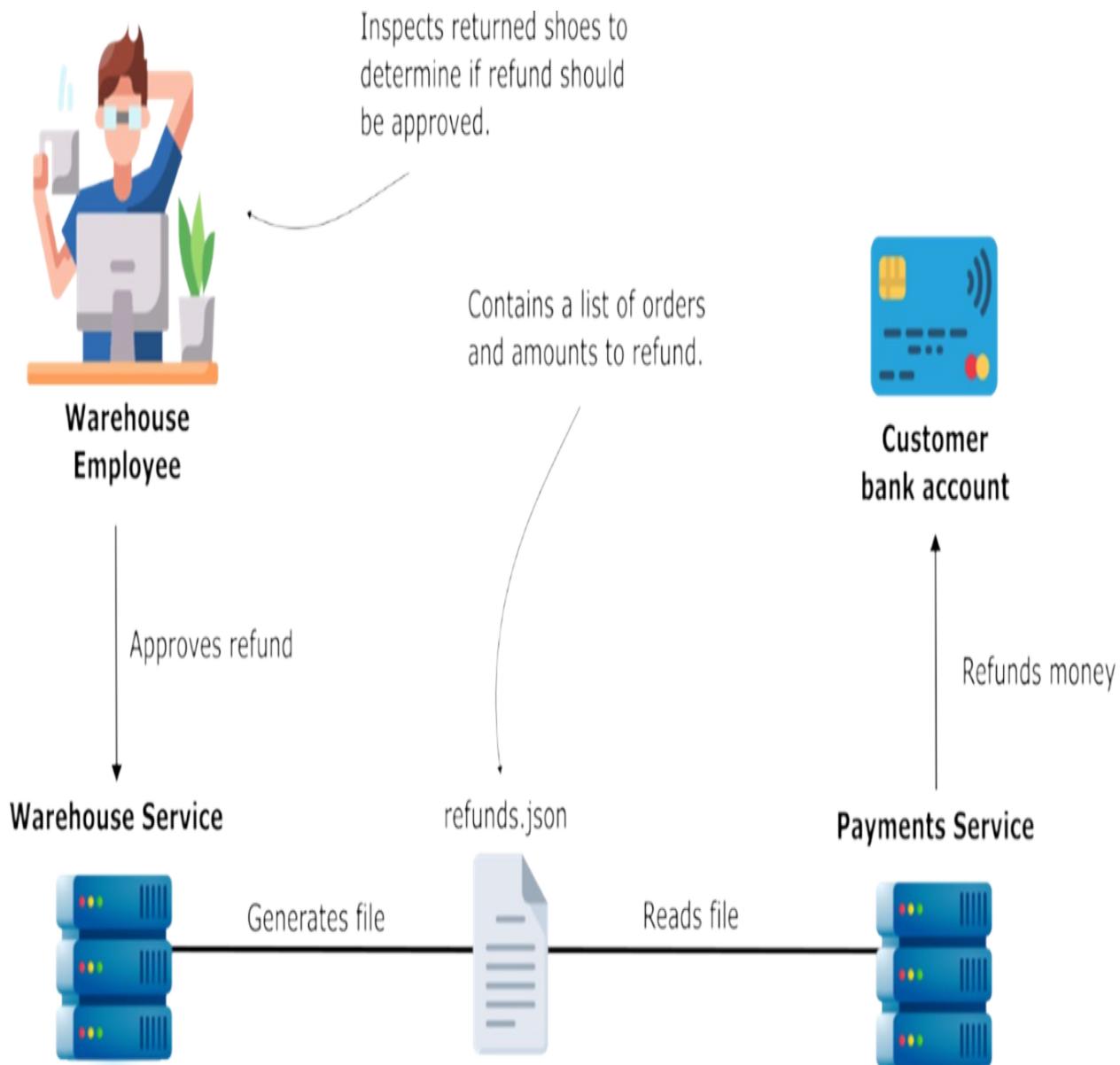
Think of it this way: ECDH is like creating a super-secret key that only you and the person you're talking to know, and AES is the lock that uses that key to keep your data safe.

We are already comfortable with AES from chapter 5, but before we dive into how elliptic curve encryption works, we first need to understand the Diffie-Hellman (DH) algorithm, which is the foundation of ECDH. It's the magic behind securely creating a shared secret key, even when communicating over an insecure network.

Let's examine some network traffic to help you understand how a Diffie-

Helman key exchange works. Let's assume the same Acme Inc. scenario that we used in the previous chapters (figure 7.2).

Figure 7.2 ACME Inc. staff inspect returned merchandise and approve refunds using the warehouse management service. Once a day the payment service requests to the warehouse management service to return a JSON array containing the credit cards and amount to refunds. The payment service issues refund to customer credit card accounts.



Diffie-Helman (DH) key agreement is a foundational algorithm used in TLS and SSH to turn an insecure communication channel into a secure communication channel by adding encryption to the channel. DH starts with

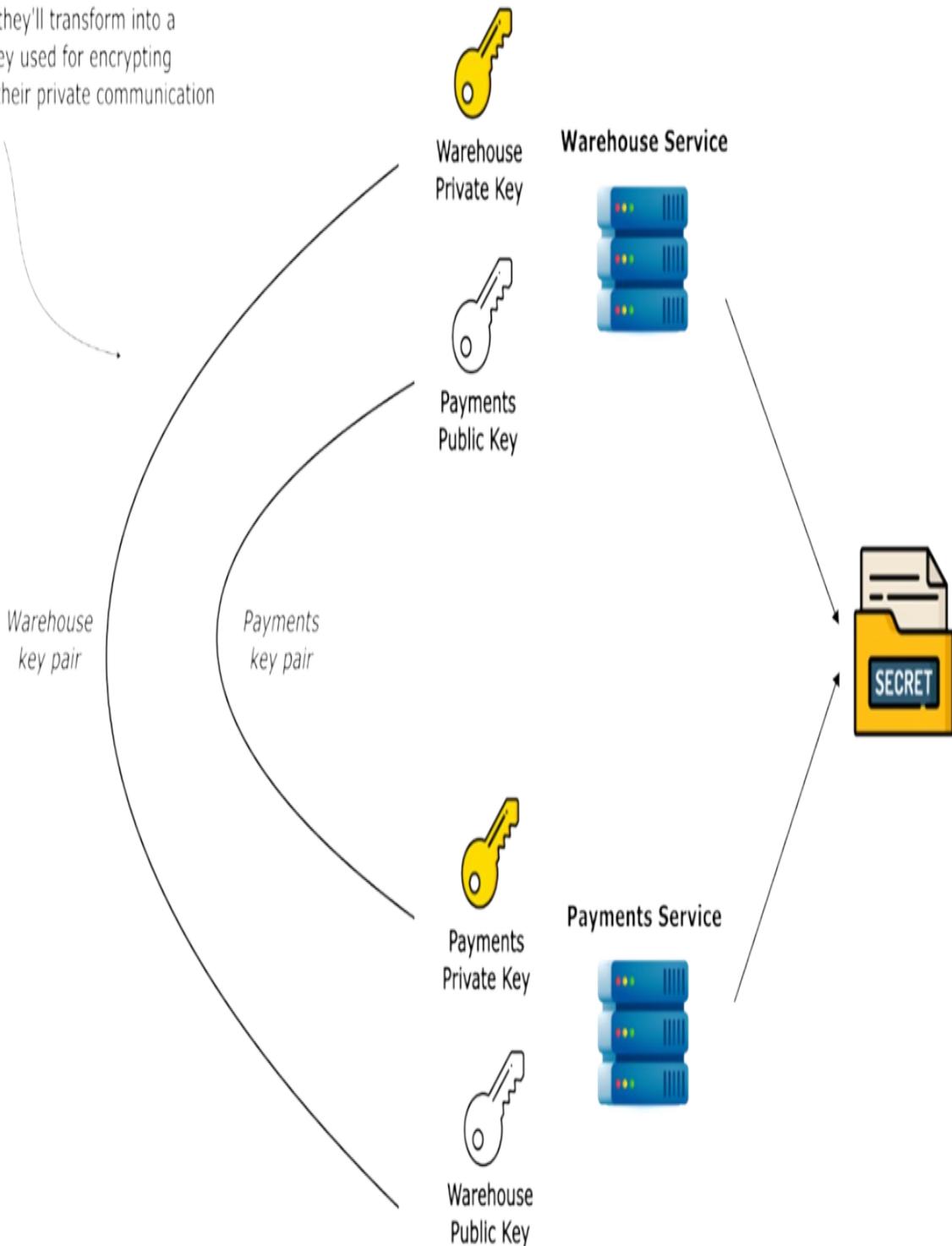
two systems exchanging their public keys and configuration parameters over an insecure network channel. Each system can then use the DH algorithm to compute the same shared secret that is never transmitted over the network. This shared secret can then be used as an AES encryption key.

Imagine you and your friend want to agree on a secret number, but you can only talk through a public chat room where everyone can see your messages. If you just send the secret directly, anyone can steal it. Instead, DH gives you a clever way to create the same secret on both sides without ever sending that secret itself.

Each of you have a public-private key pair. You exchange the public keys (figure 7.3). Private keys only remain under your eyesight. You never share these.

Figure 7.3 Diffie-Helman algorithm computes the same result from the (payment private key, warehouse public key) and (warehouse private key, payment public key) key combinations.

The services exchange the public keys in order to generate the same secret value which they'll transform into a symmetric key used for encrypting the data on their private communication channel.



You mix your friend's public key with your own private key using the DH formula. Your friend mixes your public key with their private key using the

DH formula. Because of how the math for this algorithm is designed, both sides end up with the same exact result:

```
sharedSecret = f(privateA, publicB) = f(privateB, publicA)
```

No one else can calculate this unless they know one of the private keys. This sharedSecret is just a big number. You will not use it directly but usually run it through a key-derivation function (a formula to turn it into a proper encryption key). The result is an AES key (or multiple keys) that they use to encrypt all future communication.

TIP

You will often see code and documentation referring to “Diffie-Helman Key Exchange”, which is just another name for “Diffie-Helman Key Agreement”.

Key exchange/agreement algorithms are used by secure communication protocols such as TLS and SSH to generate symmetric encryption keys to protect data in transit. As a developer you will use Diffie-Helman indirectly when you make HTTPS requests to a remote API. You don’t need to write code that calls the Diffie-Helman algorithm.

You only need decide which Diffie-Helman variant to choose when you configure a TLS connection. Part 3 of the book covers TLS from a developer’s perspective. For now, it’s important to know that there are two variants of Diffie-Helman.

- *Diffie-Helman* (DH) the original 1976 algorithm based on the discreet logarithm problem over the integers. Requires large keys that are thousands of bits long, leading to slow computation.
- *Elliptic Curve Diffie-Helman* (ECDH) a version of DH based on the discreet logarithm problem for a point on an elliptic curve. Provides better performance than original DH since it requires shorter keys. This is the variant of Diffie-Helman used in modern protocols such as TLS 1.3.

7.1.3 Exercises

1. What is the hard mathematical problem that ECC security is based on?
2. Why are not all elliptic curves safe to use in cryptography?
3. What is the difference between the original Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH)?
4. Why do developers normally not implement the Diffie-Hellman algorithm directly in their code?

7.2 Java support for elliptic curves

Java provides built-in support for working with elliptic curves, making it possible to apply this powerful cryptography in real-world applications with just a few lines of code. In this section, we'll explore two common uses of elliptic curves:

- Encryption with ECC – showing how elliptic curve cryptography can be applied to protect data.
- Signatures with ECDSA – demonstrating how elliptic curves can also be used to generate and verify digital signatures.

Together, these examples will give you a practical understanding of how ECC fits into the Java security APIs and how you can apply it in your own projects.

7.2.1 Java support for encryption with elliptic curves

Following our usual Acme Inc. example, let's assume that the development team decided to change the RSA encryption – decryption (chapter 6) with ECC. This example is also illustrated in figure 7.2. Java provides good tooling for implementing ECC encryption and decryption as well. To demonstrate this, we'll use three projects provided with the book:

- `ssfd_ch7_ex1-keygenerator` – which will generate the public/private key pair
- `ssfd_ch7_ex1-warehouse` – which will encrypt the refunds data with the public key from the pair generated with the app `ssfd_ch7_ex1-keygenerator`
- `ssfd_ch7_ex1-payments` – which will decrypt the encrypted file

produced by the Warehouse app using the private key from the pair.

The following are the steps that we'll follow to exercise the encryption and decryption with ECC:

1. Generate a new key pair with the key pair generator app.
2. Set the public key from the pair and the refunds file in the resources folder of the Warehouse app.
3. Run the Warehouse app to generate the encrypted file.
4. Take the encrypted file and the private key and store them in the resources folder of the Payments app.
5. Run the Payments app to demonstrate the decryption works.

You can also experiment by trying to decrypt with a different key or by slightly modifying the encrypted content—both attempts will fail, proving the integrity of the process. The next snippet illustrates how refunds are stored in the refunds.json file.

```
[  
  {  
    "orderId": "10001",  
    "amount": 120  
  },  
  {  
    "orderId": "10002",  
    "amount": 450  
  }  
]
```

Listing 7.1 shows the code that produces the key pair. Observe that the curve to be used needs to be specified along with a random seed value.

Listing 7.1 Generating the key pair for encryption

```
public class EcKeyPairGenerator {  
  
    private final String curveName;  
    private final SecureRandom secureRandom;  
  
    public EcKeyPairGenerator(  
        String curveName,  
        int keySize,  
        SecureRandom secureRandom)
```

```

        SecureRandom secureRandom) {

    this.curveName = curveName;
    this.secureRandom = secureRandom;
}

public KeyPair generate() throws Exception {
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");  #A
    ECGenParameterSpec ecSpec = #B
        new ECGenParameterSpec(curveName);
    kpg.initialize(ecSpec, secureRandom);  #C
    return kpg.generateKeyPair();  #D
}
}

```

Listing 7.2 shows the code used by the Warehouse project to encrypt the `refunds.json` file. While ECC can be implemented directly in plain Java, it requires quite a few more lines of code. Since the main goal of this book is not low-level implementation details, I chose to use the Bouncy Castle provider to simplify the example. (We already covered providers and the Java security API architecture in Chapter 5, so feel free to revisit that section for a refresher.)

To try it out, place the public key and the file you want to encrypt in the resources folder of the Warehouse app. Then, run the application, and you'll find the encrypted version of the file—`refunds.json.ecc`—generated in the root of the project.

Listing 7.2 Encrypting using the ECC public key from the pair

```

public final class EccEncryptor {

    public static byte[] eccEncrypt(
        byte[] data,
        PublicKey recipientPublic) throws Exception {

        if (Security.getProvider("BC") == null) {
            Security.addProvider(new BouncyCastleProvider());
        }

        Cipher ecies = Cipher.getInstance("ECIES", "BC");  #A
        ecies.init(Cipher.ENCRYPT_MODE, recipientPublic);  #B
        return ecies.doFinal(data);  #C
    }
}

```

```
    }  
}
```

Listing 7.3 shows the decryption logic in the Payments app. As before, we used the Bouncy Castle provider to keep the implementation straightforward. To try it out, copy the private key from your key pair and the encrypted file you generated with the Warehouse project into the resources folder of the Payments app. Run the application, and the decrypted file—refunds.json—will be created in the root folder of the project.

It's worth experimenting with a few scenarios to better understand how it works:

- Try using a private key from a different pair and notice that decryption fails.
- Modify the encrypted file and observe how the Payments app reacts.

Listing 7.3 Decrypting using the ECC private key from the pair

```
public class EccDecryptor {  
  
    public static byte[] eciesDecrypt(  
        byte[] ciphertext,  
        PrivateKey privateKey) throws Exception {  
  
        if (Security.getProvider("BC") == null) {  
            Security.addProvider(new BouncyCastleProvider());  
        }  
  
        Cipher cipher = Cipher.getInstance("ECIES", "BC"); #A  
        cipher.init(Cipher.DECRYPT_MODE, privateKey); #B  
        return cipher.doFinal(ciphertext); #C  
    }  
  
}
```

That wraps up our demonstration of encrypting and decrypting files with ECC in Java. You now know how to use public and private keys with the Bouncy Castle provider to secure data in practice. In the next section, we'll take things further and look at how to create and verify digital signatures using ECC.

7.2.2 Java support for signing with elliptic curves

Elliptic curves can also be used to generate digital signatures. In Chapter 6, we introduced digital signatures and demonstrated them with RSA. In this section, we'll build a similar example, but this time the application will generate signatures using the Elliptic Curve Digital Signature Algorithm (ECDSA)—an algorithm that relies on elliptic curves for creating digital signatures. You can think of ECDSA as the elliptic-curve-based alternative to RSA for digital signatures.

Listing 7.4 shows a Java implementation of ECDSA. In this example, the Warehouse project signs the encrypted data, allowing the Payments project to verify its authenticity before attempting decryption.

You notice that the identifier of the algorithm we use is SHA256withECDSA. This name comes from the fact that first, the date is hashed using SHA256 to produce a fixed output and only then the signature is produced with ECDSA.

You find this implementation in project ssfd_ch7_ex2-warehouse provided with the book.

Listing 7.4 Signing the data with a private key

```
public final class EccSigner {  
  
    public static byte[] sign(  
        byte[] data,  
        PrivateKey privateKey) throws Exception {  
  
        Signature sig = Signature.getInstance("SHA256withECDSA"); #A  
        sig.initSign(privateKey); #B  
        sig.update(data); #C  
  
        return sig.sign(); #D  
    }  
}
```

Be aware that I have generated a separate key pair for the signing process, just as we did in chapter 6 with the RSA example. The Warehouse app uses the private key of this pair for signing. The Payments app will need the public

key for verifying the signature.

We use a separate key pair for signing because the purpose of signing is different from encryption. An encryption key pair protects confidentiality, while a signing key pair ensures authenticity and integrity. Keeping these pairs separate reduces risk: even if an encryption key is compromised, the attacker still cannot create valid signatures, and vice versa.

Listing 7.5 shows the implementation in Payments app for validating the signature using the public key.

Listing 7.5 Verifying a signature with the public key

```
public class EccSignatureVerifier {  
  
    public static boolean verifyResourceSignature(  
        byte[] data,  
        byte[] sigBytes,  
        PublicKey publicKey) throws Exception {  
        Signature verifier = Signature.getInstance("SHA256withECDSA");  
        verifier.initVerify(publicKey);  #B  
        verifier.update(data);  #C  
  
        return verifier.verify(sigBytes);  #D  
    }  
}
```

That's it for our ECDSA demo in Java. We generated a dedicated signing key pair, signed the data with the Warehouse app, and verified the signature in the Payments app before attempting decryption. The flow is simple: hash → sign with the private key → send data + signature → verify with the public key. Keep the signing keys separate from encryption keys, and always verify first to avoid wasting work on tampered content.

Next, we'll explore the key differences between RSA and ECC. As we move forward, we'll revisit asymmetric encryption and digital signatures in several chapters, each time from a new perspective and in more advanced scenarios.

7.3 RSA vs. ECC

Today RSA and ECC are considered secure. Both are used in widely deployed security protocols such as SSH and TLS. As an application developer you can find high quality implementations of RSA and ECC in all popular programming language libraries. However, care must be taken to use ECC and RSA correctly by picking the right configuration, key size, and following best practices. Your corporate security standards should have up to date guidance on recommended RSA and ECC configurations.

RSA and ECC are based on mathematical problems that can be solved quickly on a quantum computer. However, quantum computers are in their infancy. At the time of writing scientists and engineers think we are still years away from practical powerful quantum computers than can break RSA and ECC. Cryptographers are developing quantum resistant public key cryptosystems that will hopefully become available before quantum computers are powerful enough to easily crack RSA and ECC.

A mathematical breakthrough that provides a fast algorithm for factoring large integers will break RSA. A breakthrough algorithm for computing the discrete logarithm for a point on an elliptic curve will break ECC.

Mathematicians do not know if fast algorithms exist or if they will ever be discovered or proven to be impossible to create. Having two mature public key cryptosystems provides us with insurance that should one cryptosystem fall to a mathematical breakthrough we have a backup that we can use.

Secure RSA keys are very large numbers, for example 4096-bits long. Performing arithmetic operations on large numbers is slow and consumes battery power on mobile devices such as phones and tablets. ECC keys are much smaller numbers than RSA keys. For example, 256-bit ECC key is thought to provide the same security as an RSA-3072 bit key.

<https://www.keylength.com> provides up to date key size recommendations using various methodologies. Overall ECC performance is better than RSA. Therefore, ECC started gaining widespread adoption in 2004 with the rise of smartphones where battery life is a priority. On the server side the performance advantage of ECC is very welcome as it lowers operating costs.

Today most cryptographers recommend ECC over RSA for new applications. Defaulting to ECC using Curve25119 is a safe choice in 2020. However, you should consult your corporate security standards to help you make the choice.

of ECC over RSA.

7.3.1 Exercises

5. What size ECC key provides security roughly equal to a 3072-bit RSA key?
6. What would break RSA, and what would break ECC?
7. Why did ECC adoption grow strongly starting around 2004?
8. Which cryptosystem do most cryptographers recommend for new applications today?

7.4 Answers to exercises

1. What is the hard mathematical problem that ECC security is based on? ECC is based on the elliptic curve discrete logarithm problem, which is extremely hard to solve.
2. Why are not all elliptic curves safe to use in cryptography? Some curves have weaknesses or can be broken. Cryptographers only approve specific curves (like Curve25519) that have been tested for safety.
3. What is the difference between the original Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH)? DH works with large integers and needs very long keys, while ECDH works with elliptic curves and gives the same security with much shorter keys and faster performance.
4. Why do developers normally not implement the Diffie-Hellman algorithm directly in their code? Because secure protocols like TLS and SSH already use DH/ECDH under the hood. Developers only need to configure which variant to use, not re-implement the math.
5. What size ECC key provides security roughly equal to a 3072-bit RSA key? A 256-bit ECC key is considered equivalent in strength to a 3072-bit RSA key.
6. What would break RSA, and what would break ECC? A fast integer-factoring algorithm would break RSA; a fast algorithm for

solving the elliptic curve discrete logarithm would break ECC.

7. Why did ECC adoption grow strongly starting around 2004?

Because smartphones and mobile devices needed strong security with small keys that use less CPU and battery.

8. Which cryptosystem do most cryptographers recommend for new applications today?

ECC, often with Curve25519, because it is efficient and secure for most modern use cases.

7.5 Summary

- Elliptic Curve Cryptography (ECC) is a modern alternative to RSA that provides the same level of security with smaller keys and better performance.
- ECC is based on the elliptic curve discrete logarithm problem, which is very hard to solve with today's computers.
- Just like RSA, ECC will be broken by powerful quantum computers, but those are not yet a practical threat.
- Different elliptic curves offer different levels of security, efficiency, and trust. Widely used safe curves include Curve25519, P-256, and secp256k1.
- Elliptic Curve Diffie-Hellman (ECDH) is used to securely generate shared symmetric keys over insecure networks, and is the variant used in modern TLS.
- Java supports ECC directly, and libraries like Bouncy Castle make it easier to encrypt and decrypt content using ECC.
- ECC is also used for digital signatures through the Elliptic Curve Digital Signature Algorithm (ECDSA). Signing and encryption must use separate key pairs to avoid weakening security.
- Digital signatures with ECC provide authenticity, integrity, and non-repudiation, while remaining more efficient than RSA for mobile devices and high-scale systems.
- Today, most cryptographers recommend ECC over RSA for new applications because of its performance and security trade-offs.

Part 3: Securing communication channels

By now you've seen the cryptographic building blocks — hashes, HMACs, AES, RSA, elliptic curves — and you know how they work in isolation. But security isn't just about the math. It's about trust. Who are you really talking to? Can you be sure that the keys you're using belong to the right person or system? And how do you keep conversations private when the network itself can't be trusted?

Part 3 connects the building blocks into systems that establish trust and protect communication over hostile networks. You'll start by learning how digital certificates and public key infrastructure (PKI) solve the "who are you talking to?" problem (chapter 8). You'll then follow the lifecycle of a certificate, from creation to validation, renewal, and even revocation. We'll also talk about how to handle self-signed certificates and set up your own certificate authority for local development (chapter 9). Finally, you'll see it all put together with TLS (chapter 10), the protocol that secures the modern internet, protects against eavesdropping and impersonation, and keeps data safe in motion.

Throughout Part 3, the Acme Inc. case study keeps us grounded in real-world software systems. You'll experiment with tools like OpenSSL, configure Spring Boot apps with certificates, and understand how an orchestrator such as Kubernetes ingress controllers and service meshes handle TLS at scale.

8 Public Key Infrastructure and X.509 Digital Certificates: know who you are talking to

This chapter covers

- Inspecting X.509 digital certificates for key fields that developers need to know
- Verifying X.509 certificates to decide if a certificate is trustworthy
- Common certificate validation failure reasons

In chapter 7, we explored how public key cryptography addresses the challenge of securely sharing a secret by using a key pair: a public key and a private key. The private key must remain confidential and securely stored, while the public key can be freely distributed without special protection.

Now, let's imagine we need to establish a secure connection between a client and a server at Acme Inc. At first glance, this might sound straightforward, but there are important details to consider. For instance, when the client and server set up their connection, they exchange public keys. This exchange happens before encryption is in place—meaning it takes place over an unsecured channel.

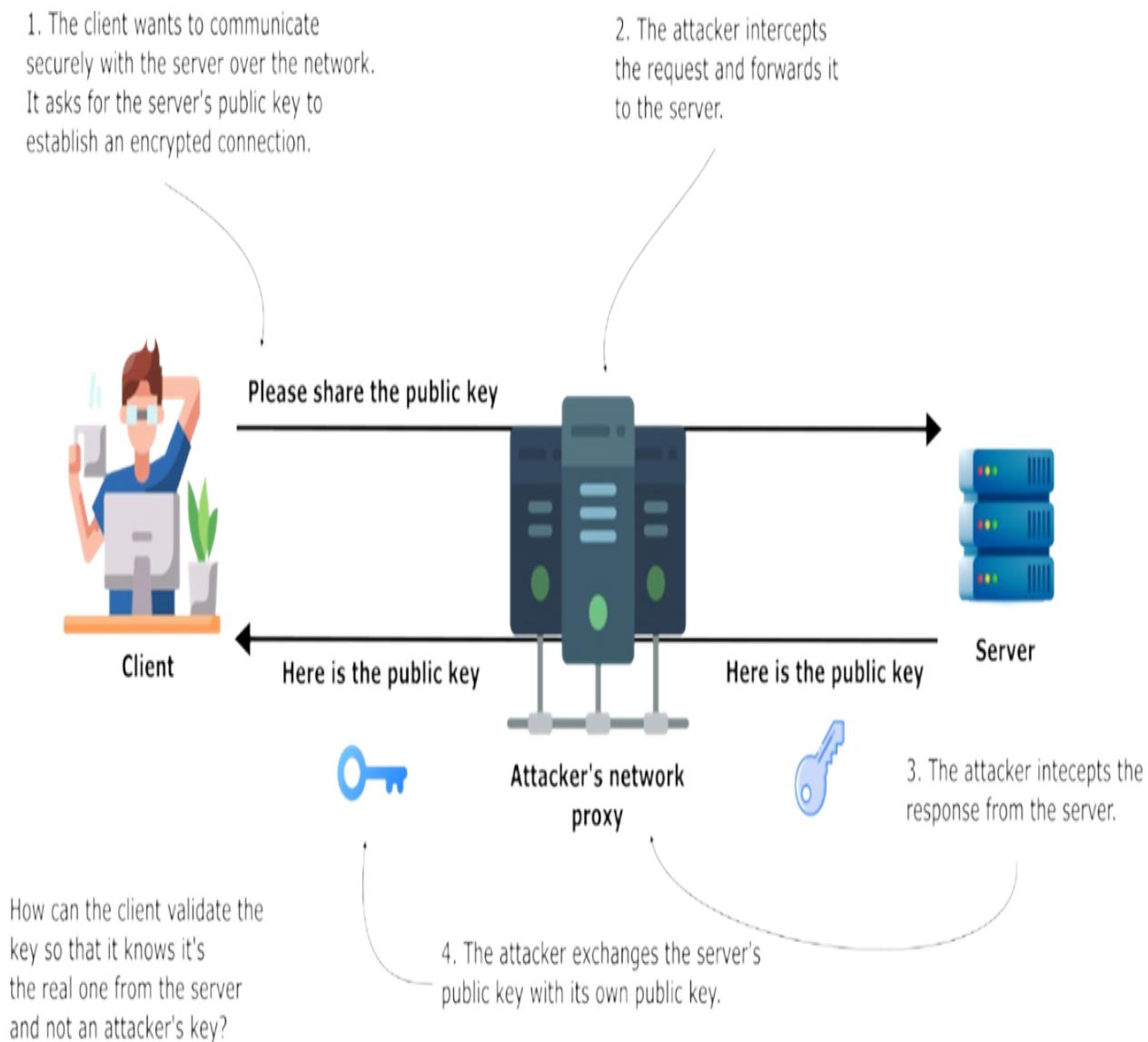
You might wonder: *Isn't that fine? After all, they're called public keys—anyone can have them!* That's true. Public keys are designed to be openly shared. However, the critical question is not whether you can receive a public key, but whose public key you are actually getting.

During this initial exchange, an attacker could intercept the communication and substitute their own public key in place of the one you expected. If that happens, the attacker could trick you into encrypting or verifying messages with their key instead of the server's legitimate key (see figure 8.1). This type of attack is known as a man-in-the-middle attack, and it highlights a key

challenge: while cryptography solves the mathematics of securing data, it doesn't by itself solve the problem of trust.

That's where certificates and Public Key Infrastructure (PKI) come in. In the next section, we'll see how digital certificates and trusted authorities help ensure that the public key you're using really belongs to the server you intend to talk to.

Figure 8.1 Public keys are exchanged over an insecure connection can be modified by a man in the middle attacker. The client needs a way to validate that the public key it received from the server has not been tampered with. Using X.509 digital certificate the client can determine who the public key belongs to.



If the client is fooled into using the attacker's public key to setup the connection, the attacker can observe and modify data exchanged over the encrypted connection. Before using a public key to setup an encrypted communication channel, a client needs a way to determine that a public key has not been modified by an attacker, and the identity of the server that the public key belongs to.

To make it easy to determine who a public key belongs to, the key is stored inside a tamper proof X.509 digital certificate. X.509 is a widely used standard for attaching metadata to a public key. A user of the public key inspects the X.509 certificate to determine who owns the public key to decide if the public key can be trusted.

The rest of the chapter will teach you everything a developer needs to know about X.509 digital certificates. We will start by learning how to inspect X.509 certificates in section 8.1 and we'll continue with an example of verifying a certificate validity in section 8.2.

TIP

Never trust a public key until you have verified who the key belongs to.

8.1 Inspecting X.509 Digital Certificates

An X.509 certificate is a data structure with standardized field names and semantics defined in [RFC 5280](<https://www.rfc-editor.org/rfc/rfc5280>). All programming languages and application servers provide libraries for working with X.509 certificates. Command line utilities that interact with remote services over the network such as curl, kubectl, git use flags to configure which X.509 certificates to use, and how to handle validation. Cloud services, Load Balancers, APIs use X.509 certificates. A solid working knowledge of X.509 certificates is essential skill for a software developer.

The best way to understand X.509 is to explore a real-world public X.509 certificate. This section is a guided tour of the github.com X.509 certificate using the openssl command line utility. We will be taking a step-by-step approach introducing key concepts of X.509 along the way, so make sure to

read this section sequentially from start to finish.

8.1.1 Inspecting X.509 certificates with OpenSSL CLI

[OpenSSL](<https://www.openssl.org>) is a popular open-source cryptography toolkit with a powerful command line interface called `openssl` that we will use to explore X.509 certificates in this chapter.

Most operating systems ship with the `openssl` CLI, however the version can be different which means that the output or command line options might be different. You can check the version of OpenSSL on your machine using the command `openssl version`. The samples used in the book were produced by OpenSSL version 3.0.13 running on Ubuntu 24.04 in a docker container. The Dockerfile used to create the container image is shown in listing 8.1.

Listing 8.1 Dockerfile with OpenSSL

```
FROM ubuntu:24.04
RUN apt-get update \
    && apt-get install -y \
        ca-certificates \
        openssl wget curl vim \
    && rm -rf /var/cache/apt/archives /var/lib/apt/lists/*
COPY examples /examples
```

If you want to follow along with the commands in the shown in this chapter, you can build the container image on your machine using the instructions at <https://github.com/securing-cloud-applications/openssl>. Alternatively, you can run the pre-built container published on the GitHub packages using the command.

```
docker run -ti ghcr.io/securing-cloud-applications/openssl:main
```

8.1.2 Downloading a Website's X.509 Digital Certificate Using OpenSSL CLI

In this chapter we'll take a public website and download and analyze the certificate it uses. The Transport Layer Security (TLS) which we'll explore in

chapter 10 is used to secure the communication over the Internet.

The *TLS* protocol uses X.509 certificates to setup an encrypted connection, which gives an opportunity to use the `openssl s_client` command to download the X.509 certificate of any public website. For example, `openssl s_client -servername github.com -connect github.com:443` is used to open a TLS connection to `github.com` and print out lots of details (100+ lines of text) including the X.509 certificate and the TLS connection configuration.

The `openssl s_client` command is interactive - it waits for user input to be typed, for example typing `GET /` and it will send an HTTP get request to the server over the TLS connection. We can avoid waiting for keyboard input by echoing an empty string and sending it via a pipe.

The command `echo | openssl s_client -servername github.com -connect github.com:443 2>/dev/null` prints out the TLS connection information without waiting for user input. However, the output contains more information than just the X.509 certificate so we filter the output of `openssl s_client` command by piping the output to `openssl x509` subcommand to extract the only the x509 certificate.

```
echo | openssl s_client -servername github.com -connect
  ➔ github.com:443 2>/dev/null | openssl x509
  ➔ > github-cert.pem
```

We combine two invocations of the `openssl` client to extract the website's certificate.

The `openssl s_client` opens a secure connection and prints out the certificate to the screen. The second invocation `openssl x509 > github-cert.pem` filters out the X.509 certificate and puts it in a file called `github-cert.pem` whose content is shown in listing 8.2.

Listing 8.2 PEM encoded X.509 `github.com` certificate at time of writing

```
-----BEGIN CERTIFICATE-----
MIIFajCCBPCgAwIBAgIQBRiaV0vox+kD4KsNk1VF3jAKBggqhkJOPQQDAzBWMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTAwLgYDVQQDEydEaWdp
Q2VydCBUTFMgSHlicmlkIEVDQyBTSEEzODQgMjAyMCBDQTEwHhcNMjIwMzE1MDAw
```

```
MDAwWhcNMjMwMzE1MjM10TU5WjBmMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2Fs
awZvcm5pYTEWMBQGA1UEBxMNU2FuIEZyYW5jaXNjbzEVMBMGA1UEChMMR2l0SHVi
LCBJbmMuMRMwEQYDVQQDEwpnaXRodWIuY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0D
AQcDQgAESrCTcYuH7GI/y3TARsjnANwnSjJLitrVrgwgRI1JlxZ1kdZQn5ltP3v7
KTtYuDdUeEu3PRx3fpDdu2cjM1yA0aOCA44wggOKMB8GA1UDIwQYMBaAFaq8CCkX
jKU5bXoOzjPHLrPt+8N6MB0GA1UDgQWBFR4qnLGcwLoFLVzsZ6LbitAh0I7hJA1
BgNVHREEHjAcggpnaXRodWIuY29tgg53d3cuZ210aHV1LmNvbTAOBgNVHQ8BAf8E
BAMCB4AwHQYDVRO1BBYwFAYIKwYBBQUAwEGCCsGAQUFBwMCMIGbBgnVHR8EgZMw
gZAwrqBEoEKGQGh0dHA6Ly9jcmwzLmRpZ21jZXJ0LmNvbS9EaWdpQ2VydFRMU0h5
YnJpZEVDQ1NIQTM4NDIwMjBDQTEtMS5jcmwwRqBEoEKGQGh0dHA6Ly9jcmw0LmRp
Z21jZXJ0LmNvbS9EaWdpQ2VydFRMU0h5YnJpZEVDQ1NIQTM4NDIwMjBDQTEtMS5j
cmwwPgYDVR0gBDcwNTAzBzZngQwBAgIwKTAAnBgrBgfFBQcCARYbaHR0cDovL3d3
dy5kaWdpY2VydC5jb20vQ1BTMIGFBgrBgfFBQcBAQR5MHcwJAYIKwYBBQUHMAGG
GGh0dHA6Ly9vY3NwLmRpZ21jZXJ0LmNvbTBpBgrBgfFBQcwAoZDaHR0cDovL2Nh
Y2VydHMuZG1naWn1cnQuY29tL0RpZ21DZXJ0VExTSH1icmlkRUNDU0hBMzg0MjAy
MENBMS0xLmNyddAJBgNVHRMEAjAAQIBfwYKKwYBBAHWeQIEAgSCAw8EggFrAwKA
dgCt9776fP8QyIudPZwePhhqtGcpXc+xDCTKhYY069yCigAAA+0i8SRAAAEawBH
MEUCIAR9cNnvYkZeKs9JE1peXwztYB2yLhtc8bB0rY2ke98nAiEAjimL8HZ7aeVE
P/DkUltwIS4c73VrG9JguRriI7gwMAdwA1zxkbv7FsV78PrUxtQsu7ticgJ1Hq
P+Eq76gdwzvWTAAAAX+0i8R7AAAEawBIMEYCIQDNckqvBhup7GpANMf0WPueytL8
u/PBaIA0bzNZeNMp0gIhAmjfEtE6AJ2fTjYCFh/BNVKk1mkTwBTavJ1GmWomQyaB
AHYAs3N3B+GEUPhjhtYFqdwRCUp5LbFnDAuH3PADDnk2pZoAAAF/jovErAAABAMA
RzBFAiEA9Uj5Ed/XjQpj/MxQRQjzG0UFQLmgW1c73nn3CJ7vskCICqHFBk1Dz7R
EHdV5Vk8bLMBW1Q6S7Ga2SbFuoVxs6zFMAoGCCqGSM49BAMDA2gAMGUCMCiVhqft
7L/stBmv1XqSRNFE/jG/AqKIBmjGTocNbuQ7kt1Cs7kRg+b3b3C9Ipu5FQIxAM7c
tGKrYDGt0pH8iF6rzbp9Q4HQXMZXkNxg+brjWxnaOVGTDNwNH7048+s/hT9bUQ==
-----END CERTIFICATE-----
```

The `github.com` X.509 certificate shown in listing 8.2 uses the Privacy Enhance Mail (PEM) format to represent the certificate as a string of ASCII characters.

PEM certificates start with a marker line containing the string `-----BEGIN CERTIFICATE-----` followed by the Base64 encoded certificate followed by a marker line `-----END CERTIFICATE-----`. PEM formatted certificates are widely used because they are easy to store in text configuration files. Non-PEM formats exist as well and we'll cover them later in this chapter.

8.1.3 Viewing The Fields of an X.509 Digital Certificate

To gain a comprehensive understanding of X.509 certificates, it's best to examine the fields of an actual X.509 certificate. This hands-on approach

reveals its structure and the associated validation rules. The openssl x509 command is a powerful tool for this purpose, as it can read PEM-encoded certificates and display all their fields in a human-readable format. The command in the next snippet generates a well-organized textual output, showcasing the fields of the X.509 certificate included in the PEM-encoded GitHub certificate from listing 8.2.

```
openssl x509 -in github-cert.pem -noout -text
```

Certificates have expiry dates so if you are following along with the sample repo, you will get different output, unless you use the certificate files form the repo.

As shown in listing 8.3, the number of fields in a X.509 can feel overwhelming, fortunately as an application developer there are only three types of fields you need to be familiar with:

- subject – which identifies the public key and its owner
- issuer – indicates who created the certificate
- validity – indicates the time range in which the certificate is valid

The remaining fields in X.509 certificates are crucial for developers building libraries to process certificates. However, we will cover them only briefly. Take a moment to review the details presented in Listing 8.3 to familiarize yourself with the structure and content of a certificate.

Listing 8.3 Contents of github.com X.509 certificate on March 5, 2023

Certificate:

Data:

Version: 3 (0x2)

Serial Number:

0c:d0:a8:be:c6:32:cf:e6:45:ec:a0:a9:b0:84:fb:1c

Signature Algorithm: ecdsa-with-SHA384

Issuer: C = US, O = DigiCert Inc, CN

↳= DigiCert TLS Hybrid ECC SHA384 2020 CA1

Validity

Not Before: Feb 14 00:00:00 2023 GMT

Not After : Mar 14 23:59:59 2024 GMT

Subject: C = US, ST = California, L = San Francisco,

```
→0 = "GitHub, Inc.", CN = github.com
  Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
        pub:
          04:a3:a4:03:46:03:df:46:51:56:cb:c9:39:ab:22:
            cd:e7:6c:59:96:7a:93:a0:fb:b9:40:1c:90:32:88:
              36:c6:09:76:9c:50:f5:55:f7:76:5e:68:20:9c:ee:
                22:ed:83:0c:15:30:10:41:44:5e:32:ac:90:a1:d5:
                  aa:f2:e5:43:b3
        ASN1 OID: prime256v1
        NIST CURVE: P-256
  X509v3 extensions:
    X509v3 Authority Key Identifier:
      0A:BC:08:29:17:8C:A5:39:6D:7A:0E:CE:33:C7:2E:B3:ED:FB:C3:
    X509v3 Subject Key Identifier:
      C7:07:27:78:85:F2:9D:33:C9:4C:5E:56:7D:5C:D6:8E:72:67:EB:
    X509v3 Subject Alternative Name:
      DNS:github.com, DNS:www.github.com
    X509v3 Key Usage: critical
      Digital Signature
    X509v3 Extended Key Usage:
      TLS Web Server Authentication, TLS Web Client Authentication
    X509v3 CRL Distribution Points:
      Full Name:
        URI:http://crl3.digicert.com
  ➔/DigiCertTLSHybridECCSHA3842020CA1-1.crl
    Full Name:
      URI:http://crl4.digicert.com
  ➔/DigiCertTLSHybridECCSHA3842020CA1-1.crl
    X509v3 Certificate Policies:
      Policy: 2.23.140.1.2.2
      CPS: http://www.digicert.com/CPS
    Authority Information Access:
      OCSP - URI:http://ocsp.digicert.com
      CA Issuers - URI:http://cacerts.digicert.com
  ➔/DigiCertTLSHybridECCSHA3842020CA1-1.crt
    X509v3 Basic Constraints:
      CA:FALSE
    CT Precertificate SCTs:
      Signed Certificate Timestamp:
        Version   : v1 (0x0)
        Log ID    : EE:CD:D0:64:D5:DB:1A:CE:C5:5C:B7:9D:B4:CD:1
                      32:87:46:7C:BC:EC:DE:C3:51:48:59:46:71:1F:B5:9B
        Timestamp : Feb 14 16:58:33.338 2023 GMT
        Extensions: none
        Signature : ecdsa-with-SHA256
```

```

30:46:02:21:00:E4:16:AE:D3:E2:2C:BA:82:9F:A9:79:
F2:4B:C6:94:52:ED:4D:E0:87:CC:50:CA:69:B1:B4:8F:
05:77:3A:94:EB:02:21:00:B5:9F:C3:F9:CB:0F:AD:D0:
60:F2:30:1B:71:05:72:12:0D:BD:65:1F:07:A9:9C:53:
4B:76:95:12:04:A6:BF:2E

Signed Certificate Timestamp:
Version : v1 (0x0)
Log ID   : 48:B0:E3:6B:DA:A6:47:34:0F:E5:6A:02:FA:9D:3
           1C:52:01:CB:56:DD:2C:81:D9:BB:BF:AB:39:D8:84:73
Timestamp : Feb 14 16:58:33.387 2023 GMT
Extensions: none
Signature : ecdsa-with-SHA256
            30:45:02:20:1E:3C:60:32:7E:20:51:F5:D6:E1:AF:7D:
            4D:F5:97:C4:48:2E:46:57:6B:86:05:37:32:4F:25:04:
            36:B1:F7:B7:02:21:00:FC:09:7E:C0:7C:03:83:26:36:
            BD:A7:5B:EB:1D:13:59:F6:62:20:8E:6D:6F:B7:0D:31:
            EB:DB:F5:11:EE:5B:D4

Signed Certificate Timestamp:
Version : v1 (0x0)
Log ID   : 3B:53:77:75:3E:2D:B9:80:4E:8B:30:5B:06:FE:4
           67:D8:4F:C3:F4:C7:BD:00:0D:2D:72:6F:E1:FA:D4:17
Timestamp : Feb 14 16:58:33.402 2023 GMT
Extensions: none
Signature : ecdsa-with-SHA256
            30:46:02:21:00:CC:E0:6B:F4:E6:74:FB:A3:92:67:21:
            53:8B:2C:0D:EB:83:F2:B0:DD:05:2D:E2:D1:C8:BE:63:
            98:4B:18:AC:36:02:21:00:EE:D2:3B:60:5A:23:08:29:
            4E:82:33:47:4A:72:A5:16:2E:46:85:13:6D:DC:DA:25:
            80:85:80:07:AA:B1:51:47

Signature Algorithm: ecdsa-with-SHA384
Signature Value:
30:64:02:30:04:dc:0d:d4:de:34:99:0a:9c:1f:a8:e1:c1:76:
5c:62:f4:04:a0:29:35:3e:c2:0d:2a:c3:71:6a:b5:f4:37:d4:
ec:0b:60:57:71:87:43:25:36:4f:c7:c2:48:d1:49:68:02:30:
56:d0:bc:c9:17:10:fb:cd:be:fe:2d:df:42:ba:c6:da:46:db:
aa:a6:67:ee:8e:88:84:81:20:85:cc:96:35:a7:b2:26:11:d6:
0c:99:9d:3c:c8:83:70:10:4b:0e:15:9b

```

8.1.4 Subject fields: Identify the public key and its owner.

Every X.509 certificate contains a public key. The owner of the public key is called the subject. The subject fields contain the public key's value and metadata about who owns the public key. If we compare a certificate to a passport, the subject fields contain the name of person, their birthdate, photo, height ... etc. The key subject fields from the github.com certificate from

listing 8.3 are extracted in listing 8.4.

Listing 8.4 Subject Fields for the github.com certificate as of March 5, 2022

```
Subject: C=US,ST=California,L=San Francisco,O="GitHub, Inc.",CN=g
Subject Public Key Info:
Public Key Algorithm: id-ecPublicKey
Public-Key: (256 bit)
pub:
04:a3:a4:03:46:03:df:46:51:56:cb:c9:39:ab:22:
cd:e7:6c:59:96:7a:93:a0:fb:b9:40:1c:90:32:88:
36:c6:09:76:9c:50:f5:55:f7:76:5e:68:20:9c:ee:
22:ed:83:0c:15:30:10:41:44:5e:32:ac:90:a1:d5:
aa:f2:e5:43:b3
ASN1 OID: prime256v1
NIST CURVE: P-256
X509v3 extensions:
X509v3 Subject Alternative Name:
DNS:github.com, DNS:www.github.com
```

The "Subject Public Key Info" section specifies the public key's value and the type of key being used. In the GitHub.com certificate shown in listing 8.4, the public key is a 256-bit elliptic curve key derived from the P-256 NIST curve. Libraries and frameworks are designed to handle the various types of public keys supported by X.509 certificates. If you need a refresher on the two most commonly used public key types—RSA and elliptic curves—refer back to chapters 6 and 7.

Two fields define the public key's owner:

- **Subject** - Identifies the main entity (person, organization, or server) the certificate was issued to.
- **Subject Alternative Name (SAN)** - Extends the Subject field by allowing multiple identifiers such as additional domain names, IP addresses, or emails.

To intuitively understand the subject fields, one must understand the historical context in which X.509 was developed.

The International Telecommunications Union (ITU) originally designed X.509 certificates with a specific use case in mind: issuing digital certificates

for individuals and storing them in a phone directory. This work was carried out in the late 1980s, prior to the invention of the World Wide Web and HTTP. As a result, the Subject field in X.509 certificates includes subfields resembling a mailing address rather than a DNS name. Take a look at the next snippet to see an example of the Subject field from the GitHub.com certificate.

```
Subject: C=US,ST=California,L=San Francisco,O="GitHub, Inc.",CN=github.com
```

The subfields are attributes of the entity that own the public key:

- C for the country
- ST for the state
- L for the locality (city, town, village)
- O for the organization
- CN for the common name

If the certificate represents a human, you might see person's name like Adib Saikali. However, when the certificate belongs to a website, it is typical to use the DNS name of the website in the subject's CN section. More subfields can be used in the subject name, but we will ignore them because they are uncommon.

The Subject Alternative Name (SAN) is a required extension field that contains a network address of the owner of the public key. Allowed network address types are DNS name, email address, IP address, or URI. The github.com certificate from listing 8.2 contains names DNS:github.com, DNS:www.github.com. Below are typical use cases for the various allowed network address types:

- DNS Name: widely used to identify server identity, it is used by clients to ensure that they are talking to the right server. For example, web browsers use the SAN DNS field to check that the certificate that is used for a TLS connection has a DNS name that matches the URL that the user typed in the browser address bar. Wild card DNS names such as *.example.com are allowed.
- IP address: can be used to identify servers. However, there is a limited

supply of IPv4 addresses so companies and cloud providers reuse IP address. If you see an IP address in X.509 certificate you can't tell who owns the IP address. It is a bad idea to use an IP address in a certificate.

- Email: can be used to create certificates to represent humans, but it is not common for people to create email-based certificates. You will likely see an X.509 certificate with an email SAN in the context of code signing using the popular sigstore.dev project.
- URI: can be used for application identity in a service-to-service call chain. In the last part of this book, when we cover SPIFEE (Secure Production Identity Framework For Everyone), you will encounter URI-based SANs.

Extracting the public key using OpenSSL CLI

As we've been discussing, each certificate has a public key. You can use the `openssl x509` command to extract the public key of a certificate. For the `github.com` certificate from listing 8.2 stored in the file `github-cert.pem` you can use the command in the next snippet to print the public key to the console.

```
openssl x509 -in github-cert.pem -pubkey -noout
```

Next snippets shows the command's result.

```
-----BEGIN PUBLIC KEY-----  
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEo6QDRgPfR1FWy8k5qyLN52xZlnqT  
oPu5QByQMog2xgl2nFD1Vfd2Xmggn04i7YMMFTAQQUReMqyQodWq8uVDsw==  
-----END PUBLIC KEY-----
```

Notice that the key is output in PEM format as a base64-encoded string with a marker line to indicate what is in the encoded string. We can decode the public key string using `openssl pkey` to process the output from the previous command. Listing 8.5 shows you how to decode the public key using `openssl` commands.

Listing 8.5 Decoding the public key

```
openssl x509 -in github-cert.pem -pubkey -noout | openssl pkey -p
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEo6QDRgPfR1FWy8k5qyLN52xZlnqT
oPu5QByQMog2xgl2nFD1Vfd2Xmgn04i7YMMFTAQQUReMqyQodWq8uVDsw==
-----END PUBLIC KEY-----
Public-Key: (256 bit)
pub:
04:a3:a4:03:46:03:df:46:51:56:cb:c9:39:ab:22:
cd:e7:6c:59:96:7a:93:a0:fb:b9:40:1c:90:32:88:
36:c6:09:76:9c:50:f5:55:f7:76:5e:68:20:9c:ee:
22:ed:83:0c:15:30:10:41:44:5e:32:ac:90:a1:d5:
aa:f2:e5:43:b3
ASN1 OID: prime256v1
NIST CURVE: P-256
```

The `github.com` certificate we decoded contains an Ecliptic Curve based public key based on the NIST P-256 curve. Recall from chapters 6 and 7 that there are two types of commonly used public key: RSA, and Elliptic Curve. An X.509 certificate can contain an RSA or Elliptic Curve key but not both.

8.1.5 Issuer Field: Identifies who created the certificate

A Certificate Authority (CA) is like an official organization that gives out digital certificates (X.509 certificates). The Issuer field in a certificate tells you which CA created it. Knowing who issued the certificate is very important, because it helps you decide whether the certificate should be trusted or not.

You can think of it like a passport: the certificate itself is the passport, and the CA is the country that issued it. If you don't trust the country, you won't trust the passport either.

Deciding whether to trust a CA is not always simple — it involves several factors that we will discuss in the next section. For now, let's focus on the X.509 fields that point to the certificate authority: Issuer and Authority Information Access (AIA).

The following snippet shows an example of the Issuer field taken directly from a certificate.

Issuer: C=US, O=DigiCert Inc, CN=DigiCert TLS Hybrid ECC SHA384 2

The issuer field is a string with a well-defined structure using the same format as the subject field of the previous section for example C=US, O=DigiCert Inc, CN=DigiCert TLS Hybrid ECC SHA384 2020 CA1 the issuer string typically contains the following fields

- C – the country where the certificate authority is located
- O - the name of the organization that owns the certificate authority
- CN – the common name of the certificate authority

In the case of the github.com certificate we can see that an organization called DigiCert Inc, based in the United States, with a certificate authority called “DigiCert TLS Hybrid ECC SHA384 2020 CA1” issued the certificate. The Issuer field contains the first breadcrumb required to validate the certificate; we will learn how to follow the breadcrumbs later in this chapter. The issuer can be extracted from an X.509 using the command as shown in the next snippet.

```
openssl x509 -in github-cert.pem -noout -issuer -ext authorityInf
```

In the following snippet you observe the result of running the previous command.

```
Issuer=C = US, O = DigiCert Inc,  
CN = DigiCert TLS Hybrid ECC SHA384 2020 CA1
```

8.1.6 Validity Fields

Just like passports X.509 certificates are valid for a specific time range. The certificate should be considered valid if the current time is within the time range set in the not before and not after fields. The validity fields specify the date range as shown in the next snippet.

Validity

```
Not Before: Feb 14 00:00:00 2023 GMT  
Not After : Mar 14 23:59:59 2024 GMT
```

Certificate lifespans can vary between a few minutes or a few years. In the

case of the `github.com` certificate we can see valid for 1 year from Feb 14, 2023, to March 14, 2024. Long-lived certificates are a security risk because it is hard to protect a private key.

Note

The longer that a certificate is valid for the more opportunity there is for the private key to be stolen, leaked, or lost.

It is a best practice to create certificates with the shortest time span reasonable for the certificate's use case. For example, code signing certificates issued by the <https://sigstore.dev> project are only valid for 10 minutes, which is long enough to sign an executable or container image in a build pipeline. Certificates issued by the Let's Encrypt certificate authority are valid for 90 days and are typically used to secure websites. We will cover some practices for certificate expiry later in this chapter.

The validity date range can be extracted using the `openssl` command as shown in the next snippet.

```
openssl x509 -in github-cert.pem -noout -dates
```

The given command produces an output similar to the one you find in the next snippet.

```
notBefore=Feb 14 00:00:00 2023 GMT
notAfter=Mar 14 23:59:59 2024 GMT
```

The `notBefore` and `notAfter` fields in the output represent the validity period of the X.509 certificate:

- `notBefore`: This is the start date and time when the certificate becomes valid. Before this date, the certificate is not considered trustworthy or usable. In this case, the certificate is valid starting from February 14, 2023, at 00:00:00 GMT.
- `notAfter`: This is the expiration date and time when the certificate ceases to be valid. After this date, the certificate is considered expired

and should no longer be used. In this case, the certificate expires on March 14, 2024, at 23:59:59 GMT.

Together, these fields define the certificate's validity period, ensuring that it can only be used within this specific timeframe.

8.1.7 X.509 Digital Certificate Encoding Formats

A common source of frustration for developers working with X.509 certificates are the various encoding formats and tools to work with them. In section 8.1.2 we looked at PEM encoded X.509 certificates, while PEM is the most used certificate format at time of writing there are other formats you might run into. No matter what data format is used to store an X.509 certificate the fields and the meaning of the fields inside the file format are the same. X.509 file formats you might encounter are:

- DER: Raw binary typically formatted using Distinguished Encoding Rules (DER) defined by the ASN.1 standard. If you see a certificate file with a .der extension it is probably contains raw bytes of a certificate. When you decode a base64 string in a PEM certificate you will end up with a DER file.
- PKCS#12: a binary cryptography file format that can store certificates and private keys, a single pkcs#12 can store multiple certificates and their associated private keys. If you see a file with a .p12 or .pfx extensions it typically a PKCS#12 formatted file. PKCS#12 is an industry standard so lots of tools understand how to process it.
- JKS: a binary format for storing a collection of certificates, and private keys used by the Java programming language. Java supports PKCS#12 files and many Java based framework can use PEM files so it is better to use PKCS#12 formatted files when working with Java. However, many online blog posts, and stack overflow answers show examples using JKS files. In the next chapter I will show you how to use PEM and PKCS#12 files with Java.

Tools like openssl can convert one certificate format to another. Stay focused on the meaning of the fields in the certificate as those fields will have the same meaning now and in the future. While overtime, you are likely to only

need to deal with PEM files.

8.1.8 Exercises

1. What are the three main X.509 certificate fields that application developers need to be familiar with?
2. What information is contained in the Subject field and in the Subject Alternative Name (SAN) extension?
3. Why is using an IP address in the SAN field generally considered a bad practice?
4. What do the Validity fields (notBefore and notAfter) represent?
5. Name two common certificate encoding formats other than PEM.

8.2 Verifying X.509 Digital Certificates

The big question we are answering in this chapter is: “Who owns a public key?”

We solved part of this by placing the public key inside an X.509 certificate. This certificate has a set of standard fields, each with a clear meaning. By looking at these fields, we can see who the owner of the public key is.

But here’s the next challenge: can we trust what the certificate says?

In everyday life, people sometimes use fake IDs — such as fake driver’s licenses, fake passports, or other forged documents. The same risk exists online: attackers can create fake digital certificates.

That’s why, in this section, we’ll focus on how to check whether a certificate is authentic and can really be trusted. Once a digital certificate is verified, you know who issued it and that it was not tampered with, but you still cannot trust it. To trust a certificate, you must trust the certificate authority that issued it. Trusting a certificate authority is a complex question, which we will explore in this section.

If you trust the certificate the authority and the certificate is valid, is the certificate safe to use? Unfortunately, it depends on the security practices of

the certificate owner. Recall from chapter 6 and 7 that a public key has a corresponding private key, if the private key is stolen, the thief can pretend to be the certificate owner.

There is no algorithmic way to determine if a private key is stolen, until the real owner of the key reports the theft to the certificate authority which adds the certificate to a Certificate Revocation List (CRL). Ideally, the owner of an X.509 certificate follows all the recommended best practices for protecting private keys. Covering all the best practices for handling private keys, is beyond the scope of this book, however, we will cover some very important patterns in upcoming chapters.

TIP

You can only trust an X.509 digital certificate if the certificate has not been tampered with, the certificate authority that issued the certificate can be trusted, and the owner of the certificate can be trusted to keep the private key private. Trusting a certificate has two components: a risk assessment, and algorithmic verification. The rest of this section teaches you the algorithmic component of certificate verification.

8.2.1 Verifying the GitHub.com X.509 Certificate

How can we be sure that an X.509 certificate is still valid and has not been damaged or changed?

Sometimes certificates can get corrupted accidentally — for example, because of hardware failures, network transmission errors, or problems when saving to disk. Other times, they may be tampered with on purpose by an attacker who wants to trick the system.

Back in chapter 4, we learned that both corruption and tampering can be detected with the help of digital signatures, which are created using hash functions. A digital signature works like a security seal: if the data changes, the seal no longer matches.

But here's the key question: which signature algorithm should you use to verify the certificate?

Every X.509 certificate includes this information inside the "Signature Algorithm" field. This field tells you which algorithm was used to sign the certificate and what value you should expect when verifying its authenticity.

Listing 8.6 shows the part of the certificate (from listing 8.2) where you find the details about the signature.

Listing 8.6 Signature section of github.com X.509 certificate on March 5, 2023

Certificate:

Data:

[Removed for brevity]

Signature Algorithm: ecdsa-with-SHA384 #A

Signature Value: #B

```
30:64:02:30:04:dc:0d:d4:de:34:99:0a:9c:1f:a8:e1:c1:76:  
5c:62:f4:04:a0:29:35:3e:c2:0d:2a:c3:71:6a:b5:f4:37:d4:  
ec:0b:60:57:71:87:43:25:36:4f:c7:c2:48:d1:49:68:02:30:  
56:d0:bc:c9:17:10:fb:cd:be:fe:2d:df:42:ba:c6:da:46:db:  
aa:a6:67:ee:8e:88:84:81:20:85:cc:96:35:a7:b2:26:11:d6:  
0c:99:9d:3c:c8:83:70:10:4b:0e:15:9b
```

The github.com certificate is signed with an Elliptic Curve Digital Signature (ECDSA) using the SHA-384 cryptographic hash function. Remember we discussed ECDSA back in chapter 7 where we also implemented a small Java app that signed data send between two apps. This information is sufficient for libraries to validate that the certificates have not been tampered with. For example, the `openssl verify` command can be used to verify a certificate as shown in the next snippet.

```
$ openssl verify github-cert.pem
```

The output of running the `openssl verify` command is shown in the next snippet.

```
C = US, ST = California, L = San Francisco,  
➥O = "GitHub, Inc.", CN = github.com  
error 20 at 0 depth lookup: unable to get local issuer certificate  
error github-cert.pem: verification failed
```

Notice that the errors output with the message `unable to get local issuer`

`certificate error` this type of error is very common, but the wording of the error message depends on the verification tool/framework used.

Understanding the root cause of this verification failure is a critical for developers. To get to the root cause you will need to understand a set of interrelated concepts:

- root certificates - self-signed certificates issued by trusted Certificate Authorities (CAs) that serve as the foundation of the certificate trust hierarchy
- intermediate certificates - certificates issued by root or higher-level intermediate CAs to create a bridge between the root certificate and end-user certificates, adding an extra layer of security
- the chain of trust - the hierarchical relationship that links an end-user certificate to a trusted root certificate through intermediate certificates, enabling the verification of the certificate's authenticity

Grab a strong cup of coffee. These are critical but challenging concepts to understand. We will explore the concepts by using OpenSSL and the `github.com` certificate, then take a step back and generalize what we have learned.

Verifying an X.509 requires validating the certificate's digital signature, which requires the public key of the certificate authority that issued the certificate. The public key of the certificate authority is stored inside an X.509 certificate, which raises the questions:

- Where to get the certificate of the certificate authority?
- How to validate the certificate of the certificate authority?

This feels like a self-referential problem, similar to the famous question: “Which came first, the chicken or the egg?” In our case, the answer is that the certificates of the certificate authorities are pre-installed as part of the operating system, browser, or language runtime. In our case, we are using an Ubuntu-based docker image.

Ubuntu ships with trusted certificate authorities in a package called `ca-certificates`. We can inspect the contents of the package using the

command `dpkg -L ca-certificates` to determine the directory location of the certificates of trusted certificate authorities as shown in listing 8.7.

Listing 8.7 Certificates Authorities pre-installed into Ubuntu 24.04

```
$ dpkg -L ca-certificates
/.
/etc
/etc/ca-certificates
/etc/ca-certificates/update.d
/etc/ssl
/etc/ssl/certs
/usr
/usr/sbin
/usr/sbin/update-ca-certificates
/usr/share
/usr/share/ca-certificates
/usr/share/ca-certificates/mozilla #A
/usr/share/ca-certificates/mozilla/ACCVRAIZ1.crt
/usr/share/ca-certificates/mozilla/AC_RAIZ_FNMT-RCM.crt
/usr/share/ca-certificates/mozilla/AC_RAIZ_FNMT-RCM_SERVIDORES_SE
/usr/share/ca-certificates/mozilla/ANF_Secure_Server_Root_CA.crt
/usr/share/ca-certificates/mozilla/Actalis_Authentication_Root_CA
/usr/share/ca-certificates/mozilla/AffirmTrust_Commercial.crt
/usr/share/ca-certificates/mozilla/AffirmTrust_Networking.crt
/usr/share/ca-certificates/mozilla/AffirmTrust_Premium.crt
/usr/share/ca-certificates/mozilla/AffirmTrust_Premium_ECC.crt
/usr/share/ca-certificates/mozilla/Amazon_Root_CA_1.crt
/usr/share/ca-certificates/mozilla/Amazon_Root_CA_2.crt
/usr/share/ca-certificates/mozilla/Amazon_Root_CA_3.crt
/usr/share/ca-certificates/mozilla/Amazon_Root_CA_4.crt
/usr/share/ca-certificates/mozilla/Atos_TrustedRoot_2011.crt
... etc extra output trimmed
```

Notice that the `/usr/share/ca-certificates/mozilla` directory contains many files where each file corresponds to a trusted certificate authority. The file names seem to contain the name of the certificate authority. Since the `github.com` certificate we are using was signed by DigiCert so we can search `/usr/share/ca-certificates/mozilla` for file names containing the string “Digi” as shown in the following snippet.

```
root@c6044a0470f5:/usr/share/ca-certificates/mozilla# ls -1 DigiC
DigiCert_Assured_ID_Root_CA.crt
```

```
DigiCert_Assured_ID_Root_G2.crt  
DigiCert_Assured_ID_Root_G3.crt  
DigiCert_Global_Root_CA.crt  
DigiCert_Global_Root_G2.crt  
DigiCert_Global_Root_G3.crt  
DigiCert_High_Assurance_EV_Root_CA.crt  
DigiCert_TLS_ECC_P384_Root_G5.crt  
DigiCert_TLS_RSA4096_Root_G5.crt  
DigiCert_Trusted_Root_G4.crt
```

Recall that the certificate authority used to sign the GitHub certificate is called “DigiCert TLS Hybrid ECC SHA384 2020 CA1,” which is not in the list above. That is the reason why we are getting the error message lookup: unable to get local issuer certificate. A web search for “DigiCert TLS Hybrid ECC SHA384 2020 CA1” leads to the DigiCert Trusted Root Authority Certificates (<https://www.digicert.com/kb/digicert-root-certificates.htm>) page on the DigiCert website (figure 8.2).

Figure 8.2 DigiCert Certificate authority with a full list of certificates of the certificate authority

The screenshot shows the DigiCert website homepage. At the top, there's a navigation bar with icons for search, refresh, and user profile. The URL bar shows 'digicert.com'. Below the header, the DigiCert logo is on the left, followed by menu links: TLS/SSL, PKI, IoT, Solutions, About, and Support. To the right are icons for globe, phone, message, and user. A main title 'DigiCert Trusted Root Authority Certificates' is centered above a sub-headline 'Download DigiCert root and intermediate certificates'. Below this is a blue banner with the text 'PROTECT YOUR SITE WITH THE WORLD'S MOST TRUSTED TLS/SSL CERTIFICATES.' and a 'BUY NOW' button.

DigiCert root certificates are widely trusted and used for issuing [TLS Certificates](#) to [DigiCert](#) customers—including educational, financial institutions, and government entities worldwide.

DigiCert strongly recommends including each of these roots in all applications and hardware that support X.509 certificate functionality, including Internet browsers, email clients, VPN clients, mobile devices, operating systems, etc.

DigiCert Customers: If you are looking for your certificate's intermediate root, please download it from inside your [DigiCert account](#) or contact your account manager or [DigiCert Support](#).

Additional resources

- For DigiCert community root and intermediate certificates, visit [DigiCert Community Root and Authority Certificates](#).
- For the QuoVadis brand root CA certificates, visit [QuoVadis CA Certificate Download](#)

Not finding the DigiCert CA certificates you are searching for?

DigiCert discloses all of its public root and intermediate certificates on [Common CA Database](#). If you do not see the root certificate or cross-certificate that you need, have any questions, or would like to be added to our supported applications list, please contact us at roots@digicert.com.

* [Root Certificates](#)

* [Intermediate Certificates](#)

* [Cross Signed Certificates](#)

Root Certificates

* [GS root certificates](#)

* [Other root certificates](#)

G5 root certificates

Name	Details
DigiCert TLS ECC P384 Root G5 Download PEM Download DER/CRT	Valid until: 14/Jan/2046 Serial #: 09:E0:93:65:AC:F7:D9:C8:B9:3E:1C:0B:04:2A:2E:F3 SHA1 Fingerprint: 17:F3:DE:5E:9F:0F:19:E9:8E:F6:1F:32:26:6E:20:C4:07:AE:30:EE SHA256 Fingerprint: 01:8E:13:F0:77:25:32:CF:80:9B:D1:B1:72:81:86:72:83:FC:48:C6:E1:3B:E9:C6:98:12:85:4A:49:0C:1B:05 Demo Sites for Root: Active Certificate expired revoked

On the DigiCert Trusted Root Authority Certificates page, we search for the string “DigiCert TLS Hybrid ECC SHA384 2020 CA1” to find the details of the certificate authority's cert (figure 8.3).

Figure 8.3 Details Links for the certificate of the DigiCert TLS Hybrid ECC SHA384 2020 CA1 certificate authority



Using the data on the page and the `wget` CLI tool we download the X.509 certificate of the certificate authority that issued the certificate for githhub.com using the command.

```
 wget https://cacerts.digicert.com/
 ➔ DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem
```

Listing 8.8 shows you the result of inspecting the downloaded certificate with `openssl` commands.

Listing 8.8 DigiCert TLS Hybrid ECC SHA384 2020 CA1 X.509 certificate

```
$ openssl x509 -text -noout -in
 ➔ DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:f2:f3:5c:87:a8:77:af:7a:ef:e9:47:99:35:25:bd
        Signature Algorithm: sha384WithRSAEncryption
        Issuer: C = US, O = DigiCert Inc,
 ➔ OU = www.digicert.com, CN = DigiCert Global Root CA #A
        Validity #B
            Not Before: Apr 14 00:00:00 2021 GMT
```

Not After : Apr 13 23:59:59 2031 GMT
Subject: C = US, O = DigiCert Inc,
→CN = DigiCert TLS Hybrid ECC SHA384 2020 CA1 #C
Subject Public Key Info:
 Public Key Algorithm: id-ecPublicKey
 Public-Key: (384 bit)
 pub:
 04:c1:1b:c6:9a:5b:98:d9:a4:29:a0:e9:d4:04:b5:
 db:eb:a6:b2:6c:55:c0:ff:ed:98:c6:49:2f:06:27:
 51:cb:bf:70:c1:05:7a:c3:b1:9d:87:89:ba:ad:b4:
 13:17:c9:a8:b4:83:c8:b8:90:d1:cc:74:35:36:3c:
 83:72:b0:b5:d0:f7:22:69:c8:f1:80:c4:7b:40:8f:
 cf:68:87:26:5c:39:89:f1:4d:91:4d:da:89:8b:e4:
 03:c3:43:e5:bf:2f:73
 ASN1 OID: secp384r1
 NIST CURVE: P-384
X509v3 extensions:
 X509v3 Basic Constraints: critical
 CA:TRUE, pathlen:0
 X509v3 Subject Key Identifier:
 0A:BC:08:29:17:8C:A5:39:6D:7A:0E:CE:33:C7:2E:B3:E
 X509v3 Authority Key Identifier:
 03:DE:50:35:56:D1:4C:BB:66:F0:A3:E2:1B:1B:C3:97:B
 X509v3 Key Usage: critical
 Digital Signature, Certificate Sign, CRL Sign
 X509v3 Extended Key Usage:
 TLS Web Server Authentication, TLS Web Client Aut
 Authority Information Access:
 OCSP - URI:http://ocsp.digicert.com
 CA Issuers -
→URI:http://cacerts.digicert.com/
→DigiCertGlobalRootCA.crt
 X509v3 CRL Distribution Points:
 Full Name:
 URI:http://crl3.digicert.com/DigiCertGlobalRoot
 X509v3 Certificate Policies:
 Policy: 2.16.840.1.114412.2.1
 Policy: 2.23.140.1.1
 Policy: 2.23.140.1.2.1
 Policy: 2.23.140.1.2.2
 Policy: 2.23.140.1.2.3
Signature Algorithm: sha384WithRSAEncryption
Signature Value:
 47:59:81:7f:d4:1b:1f:b0:71:f6:98:5d:18:ba:98:47:98:b0:
 7e:76:2b:ea:ff:1a:8b:ac:26:b3:42:8d:31:e6:4a:e8:19:d0:
 ef:da:14:e7:d7:14:92:a1:92:f2:a7:2e:2d:af:fb:1d:f6:fb:
 53:b0:8a:3f:fc:d8:16:0a:e9:b0:2e:b6:a5:0b:18:90:35:26:

```
a2:da:f6:a8:b7:32:fc:95:23:4b:c6:45:b9:c4:cf:e4:7c:ee:  
e6:c9:f8:90:bd:72:e3:99:c3:1d:0b:05:7c:6a:97:6d:b2:ab:  
02:36:d8:c2:bc:2c:01:92:3f:04:a3:8b:75:11:c7:b9:29:bc:  
11:d0:86:ba:92:bc:26:f9:65:c8:37:cd:26:f6:86:13:0c:04:  
aa:89:e5:78:b1:c1:4e:79:bc:76:a3:0b:51:e4:c5:d0:9e:6a:  
fe:1a:2c:56:ae:06:36:27:a3:73:1c:08:7d:93:32:d0:c2:44:  
19:da:8d:f4:0e:7b:1d:28:03:2b:09:8a:76:ca:77:dc:87:7a:  
ac:7b:52:26:55:a7:72:0f:9d:d2:88:4f:fe:b1:21:c5:1a:a1:  
aa:39:f5:56:db:c2:84:c4:35:1f:70:da:bb:46:f0:86:bf:64:  
00:c4:3e:f7:9f:46:1b:9d:23:05:b9:7d:b3:4f:0f:a9:45:3a:  
e3:74:30:98
```

We downloaded a certificate from the Internet, where hackers like to do malicious things. How do we know that this certificate was not tampered with in some way? We need to verify it, using `openssl`. The following snippet shows you how to use `openssl` to verify the certificate as shown in the next snippet.

```
openssl verify DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem
```

The following snippet shows us the response for the given command. The certificate is valid.

```
DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem: OK
```

Success! We can trust that `DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` has not been tampered with.

But how did `openssl` do the verification? OpenSSL checked the digital signature of `DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` using the public key of the certificate authority that issued `DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` certificate.

What is that certificate authority that issued a certificate for another certificate authority? We can discover the answer by inspecting `DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` using `openssl` as shown by the next snippet.

```
openssl x509 -noout -issuer -in DigiCertTLSHybridECCSHA3842020CA1
```

In the following snippet you observe the details from the certificate provided as a result of running the `openssl x509` command.

```
issuer=C = US, O = DigiCert Inc, OU = www.digicert.com,  
CN = DigiCert Global Root CA
```

Looking at the CN field it looks like “DigiCert Global Root CA” issued the

`DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` certificate. Recall that the Ubuntu operating system ships with certificate authority files installed in the `/usr/share/ca-certificates/Mozilla` directory containing `DigiCert_Global_Root_CA.crt`. We can observe that by inspecting `DigiCert_Global_Root_CA.crt` using `openssl` as shown in the next snippet.

```
openssl x509 -noout -subject -in /usr/share/  
ca-certificates/mozilla/DigiCert_Global_Root_CA.crt
```

The next snippet shows the result of running the command.

```
subject=C = US, O = DigiCert Inc, OU = www.digicert.com,  
CN = DigiCert Global Root CA
```

Notice that the subject of the `DigiCert_Global_Root_CA.crt` certificates matches the issuer of the `DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem` certificate. We can ask the `openssl` to print out the steps it is taking to verify the certificate using the `-show_chain` option.

The following snippet shows the command you need to run to show the certificate chain for `github.com`

```
openssl verify -show_chain -CAfile  
DigiCertTLSHybridECCSHA3842020CA1-1.crt.pem github-cert.pem
```

Listing 8.9 shows the output after running the command.

Listing 8.9 The certificate chain

```
github-cert.pem: OK #A
Chain:
depth=0: C = US, ST = California, L = San Francisco, O = "GitHub,
➥CN = github.com (untrusted) #B
depth=1: C = US, O = DigiCert Inc,
➥CN = DigiCert TLS Hybrid ECC SHA384 2020 CA1 #C
depth=2: C = US, O = DigiCert Inc, OU = www.digicert.com,
➥CN = DigiCert Global Root CA #D
```

We asked openssl to verify that the github.com certificate was not tampered with. openssl validated the digital signature in the github.com certificate using the “DigiCert TLS Hybrid ECC SHA384 2020 CA1” X.509 certificate, which was in turn validated using the “DigiCert Global Root CA”, which is trusted because it is pre-installed in Ubuntu 24.04 OS. Three certificates were involved, where each certificate played one of the following roles:

- End entity – this is the github.com certificate used to establish a secure connection to github
- Intermediate certificate authority – The certificate used to issue the github.com certificate.
- Root certificate authority – the certificate used to issue the intermediate certificate authority’s certificate. It is trusted because it is installed by the operating system.

Visualizing the three certificates and relationship between them.

Figure 8.4 How trust is established by following the certificate chain to a trusted root certificate authority. Hint read this diagram from the bottom to the top.

This is a certificate authority that can issue certificates, including certificates for other certificate authorities.

Root of trust: trusted because it is installed into the operating system by the operating system vendor.

25 year lifespan

The issuer and the subject are the same since this is a root certificate.

X.509 Root Certificate Authority Certificate	
X509v3 Key Usage: critical Digital Signature, Certificate Sign, CRL Sign X509v3 Basic Constraints: critical → CA: TRUE	Usage Restrictions
Validity Not Before: Nov 10 00:00:00 2006 GMT Not After: Nov 10 00:00:00 2031 GMT	Validity Period
Issuer C=US,O=DigiCert Inc, OU=www.digicert.com,CN=DigiCert Global Root CA	Issuing Authority
Subject C=US,O=DigiCert Inc, OU=www.digicert.com,CN=DigiCert Global Root CA	Owner Details

Trusted if issuer is trusted.

Validity

Not Before: Apr 13 00:00:00 2021 GMT
Not After: Apr 14 00:00:00 2031 GMT

Subject

C=US,O=DigiCert Inc,CN=DigiCert TLS Hybrid ECC SHA384 2020 CA1

Trusted if issuer is trusted.

1 years lifespan

Domain that owns the certificate

Can be used only to make TLS connections and to check digital signatures. It is not a certificate authority.

X.509 Intermediate Certificate Authority Certificate

Issuer C=US,O=DigiCert Inc, OU=www.digicert.com,CN=DigiCert Global Root CA	Issuing Authority
X509v3 Basic Constraints: critical CA:TRUE, pathlen: 0 X509v3 Key Usage: critical Digital Signature,Certificate Sign, CRL Sign X509v3 Extended Key Usage: TLS Web Server Authentication, TLS Web Client Authentication	Usage Restrictions
Validity Not Before: Apr 13 00:00:00 2021 GMT Not After: Apr 14 00:00:00 2031 GMT	Validity Period
Subject C=US,O=DigiCert Inc,CN=DigiCert TLS Hybrid ECC SHA384 2020 CA1	Owner Details

Trusted if issuer is trusted.

It is a certificate authority but it can only issue certificates for end use.

10 years lifespan

X.509 End Entity TLS Certificate

Issuer C=US,O=DigiCert Inc,CN=DigiCert TLS Hybrid ECC SHA384 2020 CA1	Issuing Authority
Validity Not Before: Feb 14 00:00:00 2023 GMT Not After: Feb 14 00:00:00 2024 GMT	Validity Period
X509v3 Subject Alternative Name: → DNS: github.com, DNS:www.github.com Subject: C=US, ST=California, L=San Francisco, O="GitHub, Inc.", CN=github.com	Owner Details
X509v3 Basic Constraints: critical CA:FALSE X509v3 Key Usage: critical Digital Signature X509v3 Extended Key Usage: TLS Web Server Authentication, TLS Web Client Authentication	Usage Restrictions

TIP

The certificate chain of trust is a critical. Wrapping your head around this concept can be challenging. If you are struggling, try using the commands you have seen in this chapter, to download the certificate for amazon.com and replicate what you have seen so far. Hands-on engagement with concepts can help clarify the ideas. In this section, we dealt with the details of the github.com certificate. In the next section, we will dive deeper and look more in depth at certificate authorities and chain of trust.

I hope this chapter just opened your appetite and now you have more questions about the subject such as:

- Why does the root certificate have 25-year lifespan, the intermediate certificate has 10-year lifespan, and the end entity certificate has a 1-year lifespan?
- What is the difference between an intermediate certificate authority and a root certificate authority?
- Why are intermediate certificate authorities required?
- How exactly do certificate authorities issue certificates?

In the next chapter, we will focus on these questions;

8.2.2 Exercises

6. What role does the Signature Algorithm field play in an X.509 certificate?
7. What is the purpose of a root certificate authority versus an intermediate certificate authority?
8. Why do operating systems and browsers ship with a collection of pre-installed root certificates?
9. In the certificate chain of trust, what are the three types of certificates involved?
10. Why can't we always know if a private key has been stolen, and how is revocation handled when this happens?

8.3 Answers to exercises

1. What are the three main X.509 certificate fields that application developers need to be familiar with?
The three key fields are Subject, Issuer, and Validity.
2. What information is contained in the Subject field and in the Subject Alternative Name (SAN) extension?
The Subject identifies the entity the certificate was issued to (organization, domain, or individual). The SAN field extends this by listing additional identifiers such as DNS names, IPs, URIs, or emails.
3. Why is using an IP address in the SAN field generally considered a bad practice?
Because IP addresses can be reassigned and reused, an IP in a certificate does not reliably identify the server owner.
4. What do the Validity fields (notBefore and notAfter) represent?
The notBefore date marks when the certificate becomes valid, and the notAfter date marks when it expires.
5. Name two common certificate encoding formats other than PEM.
Two other common formats are DER (binary ASN.1 encoding) and PKCS#12 (often with .p12 or .pfx extensions).
6. What role does the Signature Algorithm field play in an X.509 certificate?
It specifies the cryptographic algorithm used to sign the certificate (e.g., ECDSA with SHA-384) and is required to verify authenticity.
7. What is the purpose of a root certificate authority versus an intermediate certificate authority?
A root CA is a self-signed, widely trusted authority. Intermediate CAs issue end-entity certificates and create a safer trust hierarchy so that root certificates are not used directly.
8. Why do operating systems and browsers ship with a collection of pre-installed root certificates?
So that applications and users can instantly verify certificates without having to fetch and manually trust each CA. This collection is called the trust store.
9. In the certificate chain of trust, what are the three types of certificates involved?
The three types are: end-entity certificate (e.g., github.com),

intermediate certificate, and root certificate authority.

10. Why can't we always know if a private key has been stolen, and how is revocation handled when this happens?

Because key theft leaves no technical trace. Revocation is handled through Certificate Revocation Lists (CRLs) or the Online Certificate Status Protocol (OCSP) after the owner reports the theft.

8.4 Summary

- X.509 certificates contain a public key and metadata about who owns the key and who issued it.
- The three key certificate fields developers need to know are Subject (owner identity), Issuer (certificate authority), and Validity (time range).
- Subject Alternative Name (SAN) extension lists network addresses like DNS names, emails, IP addresses, or URIs.
- Certificate authorities (CAs) digitally sign certificates to prove authenticity and prevent tampering.
- Root CAs are self-signed and pre-installed in operating systems, while intermediate CAs create a safer trust hierarchy.
- Certificate verification follows a chain of trust from end-entity certificate to intermediate CA to trusted root CA.
- Common certificate formats include PEM (text-based), DER (binary), and PKCS#12 (can store multiple certificates and keys).
- Certificate validation checks digital signatures, validity dates, and that the CA is trusted.
- Domain validation proves certificate requesters control the domain through email, HTTP-01, or DNS-01 challenges.
- Certificate revocation lists (CRLs) track stolen or compromised certificates that should no longer be trusted.
- Private certificate authorities can be created for local development to avoid self-signed certificate warnings.
- OpenSSL command-line tools can inspect, verify, and create certificates for testing and development.

9 Working with X.509 Certificates: Lifecycle and Self-Signing

This chapter covers

- Optimal way to do local development using a laptop scoped certificate authority
- Creating X.509 digital certificates with the Automated Certificate Management Environment (ACME) protocol
- Renewing X.509 digital certificates using ACME

So far in chapter 8, we've looked at what's inside an X.509 certificate. Basically, it's got two things: a public key, and a bunch of extra info about that key. We poked around some of the more important bits like who the certificate belongs to (the subject), who gave it out (the issuer), how long it's good for, and a few rules and restrictions.

You've also learned how to check if a certificate is legit: you follow the trail of who issued it, step by step, all the way up to a trusted root certificate. Kinda like checking if someone's ID was signed by someone you actually trust.

So yes, we know who gives out these certificates. It's the certificate authorities. But how exactly do they do it? Good question. Let's find out! In this chapter we refine our understanding of certificate authorities by learning the certificate lifecycle from issuance to expiry or revocation and usage of self-signed certificates. We will continue the exploration using the `openssl` CLI to build up our certificate authority. While developers do not need to create and manage their own certificate authorities, the key to developer certificate superpowers lies in understanding the complex ideas that we will explore in this chapter. So, grab another cup of coffee, and let's dive in.

9.1 Certificate Lifecycle: Issuance to revocation

Let's start by taking a closer look at how certificates are born, live their life, and eventually retire or get kicked out. We'll walk through the whole journey, step by step.

First up, in section 9.1.1, we'll talk about *creating a key pair*. Think of this as making a lock and key: one part (the private key) stays secret with you, while the other part (the public key) is something you can safely share with the world. You'll need both to make the certificate magic happen.

Next, in section 9.1.2, we'll look at how you *create a Certificate Signing Request (CSR)*. This is basically you saying, “*Hey, here’s my public key and some info about me, can I please get a certificate?*” It’s like filling out a form and sending it to the certificate authority (CA) with your digital signature on it.

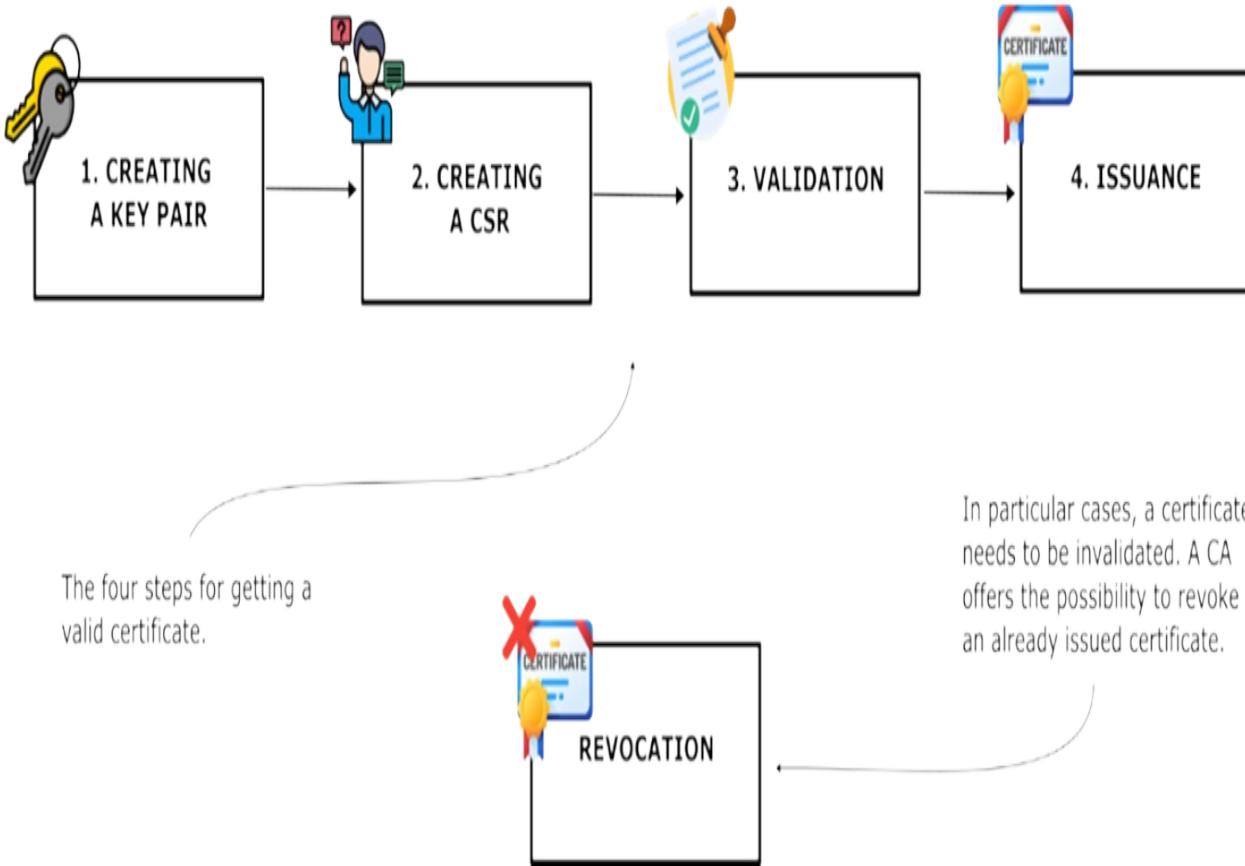
Then, in section 9.1.3, we dive into *validation*. This is the CA’s way of saying, “*Wait a sec, are you really who you say you are?*” Depending on the type of certificate, this might be a quick check or a full-on investigation. No trench coats or magnifying glasses involved (usually), but still serious business.

In section 9.1.4, we get to the good part: *issuance*. If everything checks out, the CA creates your certificate and hands it over. Congrats! You’re now officially certified. You can show off your public key proudly, and people can trust that it really belongs to you.

Finally, in section 9.1.5, we cover *revocation*: the sad but important part of the story. Sometimes, things go wrong. Maybe your private key gets leaked, or you’re no longer supposed to have the certificate. In that case, the certificate needs to be revoked, basically, put on a digital “do not trust” list so others know not to use it anymore.

Figure 9.1 summarizes the steps.

Figure 9.1 The certificate lifecycle. This diagram walks you through the four main steps to getting a valid digital certificate: generating a key pair, creating a Certificate Signing Request (CSR), having it validated by a Certificate Authority (CA), and receiving the issued certificate.

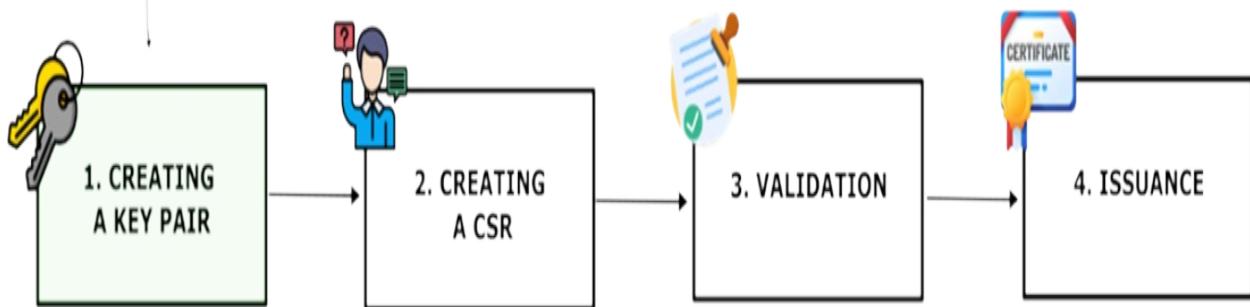


9.1.1 Creating a keypair

A X.509 certificate contains a public key, which means there must be a private key corresponding to the public key. The first step in creating a certificate is to create a public / private key pair (figure 9.2).

Figure 9.2 Step 1: It all begins with a key pair. The lifecycle of a certificate kicks off with generating a key pair: one public key, one private key. This pair becomes the foundation for your Certificate Signing Request (CSR), which you'll send to the Certificate Authority (CA) to prove you're legit.

The first step is creating a key pair.
The key pair will be used in the creation request you send to the Certificate Authority.



Using the `openssl genpkey` command as shown in the next snippet we can accomplish this task. The next snippet shows the command you can use to achieve the desired result.

```
$ openssl genpkey -algorithm Ed25519 -outform PEM -out private_ke
```

We set three parameters. The `-algorithm` parameter sets the type of public key encryption algorithm. In this case we are choosing to use the elliptic curve called Ed25519 and output the key to a file called `private_key.pem` (in the PEM file format). Ed25519 keys are very small.

Remember elliptic curves? We talked about them in chapter 7. No worries if they sound a bit fuzzy, they're basically a clever bit of math used to create secure keys with much smaller sizes than the old-school RSA ones. Instead of relying on giant prime numbers, elliptic curve algorithms use points on a special kind of curve (yep, an actual curve from math class) to do the job.

When you see names like Ed25519 or secp256r1, those are just different types of elliptic curves. Each curve with its own specific shape and properties. Ed25519 in particular is known for being fast, secure, and compact, which is why we're using it here. So when you set the `-algorithm` parameter to something like Ed25519, you're saying, “Hey, let's use that tiny but mighty curve for our key.”

To show the file's content use the cat command as shown in the next snippet.

```
$ cat private_key.pem
```

You will get the output as shown in the next snippet.

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEIAgbK6DGz3JC02saDwhzhdxBr/GjTVDTIw0Po7vEN//d
-----END PRIVATE KEY-----
```

Did you expect the genkey command to generate a public key? In fact, the output file contains a private key. This is because every public key requires a corresponding private key. While it is possible to store these keys in separate files, managing them separately can be cumbersome. For practical reasons, it is often more efficient for the private key owner to store the key pair in a single file. You can view the file's contents using the appropriate command shown in the next snippet.

```
$ openssl pkey -in private_key.pem -text -noout
```

Listing 9.1 shows the public/private key pair that was generated as a result of running the discussed command.

Listing 9.1 The generated public/private key pair

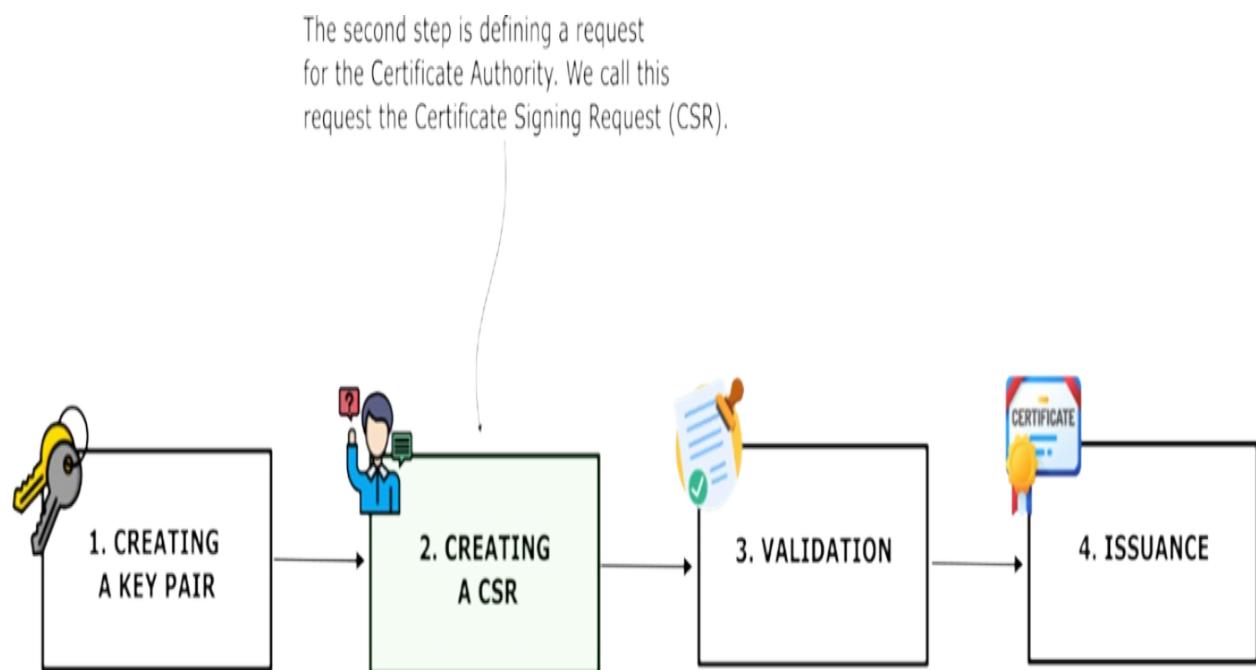
```
ED25519 Private-Key:
priv: #A
    08:1b:2b:a0:c6:cf:72:42:d3:6b:1a:0f:08:73:85:
    dc:41:af:f1:a3:4d:50:d3:23:0d:0f:a3:bb:c4:37:
    ff:dd
pub: #B
    0a:76:42:f1:fa:55:57:a0:c2:b9:de:79:d3:0c:4a:
    50:b8:07:bf:78:ee:0f:45:99:a5:ec:cd:98:07:50:
    c1:33
```

Notice that the output contains both the private and public key.

9.1.2 Creating a Certificate Signing Request (CSR)

As we've seen in chapter 8, certificates are issued by a certificate authority (CA). This means we need to send the public key and metadata about the public key to a certificate authority so that the certificate authority can issue a certificate (figure 9.3).

Figure 9.3 Step 2: Submitting a CSR. Once you've got your key pair, the next move is to create a Certificate Signing Request (CSR). This is like filling out a form saying, "Hey CA, here's who I am, and here's my public key, can I have a certificate, please?" The CA will use this info to decide whether you're worthy.



A Certificate Signing Request (CSR) is a file containing a public key and metadata about the key sent by the owner of the public key to a certificate authority. We can create a CSR using `openssl req` command to generate a CSR.

```
$ openssl req -new -key private_key.pem -out request.csr
```

Listing 9.2 shows the interactive input form that appears in response to the `openssl req` command demonstrated in the previous snippet. The user is prompted to provide details required for creating the certificate. These prompts typically include fields such as the country name, state or province, organization name, email address, and other relevant information. Each input

contributes to the certificate's distinguished name (DN), which uniquely identifies the entity associated with the certificate.

Listing 9.2 Creating a Certificate Signing Request (CSR)

You are about to be asked to enter information that will be incorporated into your certificate request.

What you are about to enter is what is called a Distinguished Name.

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

----- #A

Country Name (2 letter code) [AU]:CA

State or Province Name (full name) [Some-State]:ON

Locality Name (eg, city) []:Toronto

Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example

Organizational Unit Name (eg, section) []:

Common Name (e.g. server FQDN or YOUR name) []:example.com

Email Address []:admin@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request

A challenge password []:

An optional company name []:

To generate the request the openssl req command prompts the user for the details of the subject that should be associated with the public key. It generates a file called request.csr that contains the public key and the metadata about the key collected from the answers. The next snippet presents the content of the certificate request file.

```
-----BEGIN CERTIFICATE REQUEST-----
MIH7MIGuAgEAMHsxCzAJBgNVBAYTAKNBMQswCQYDVQQIDAjPTjEQMA4GA1UEBwwH
VG9yb250bzEVMBMGA1UECgwMRXhhbXBsZSBJbmMuMRQwEgYDVQQDDAtleGFtcGx1
LmNvbTEgMB4GCSqGSIb3DQEJARYRYWRtaW5AZXhhbXBsZS5jb20wKjAFBgMrZXAD
IQAKdkLx+1VXoMK53nnTDEpQuAe/e04PRZml7M2YB1DBM6AAMAUGAytlcANBAFGD
OpE0iql2WESusN6kPhtQvWQMb/+tbm/zl+o+GfIC8w+3ihzBaduzjzWebehKL8h6
zvLfbz7mgIcnfxpkxAE=
-----END CERTIFICATE REQUEST-----
```

You can print out the details of the CSR using the command presented in the next snippet.

```
$ openssl req -in request.csr -text -noout
```

In listing 9.3 you see the certificate request details as a response to the `openssl req` command given in the previous snippet.

Listing 9.3 The certificate request details

Certificate Request:

 Data:

```
    Version: 1 (0x0)
    Subject: C = CA, ST = ON, L = Toronto,
    ➥O = Example Inc., CN = example.com,
    ➥emailAddress = admin@example.com
```

 Subject Public Key Info:

 Public Key Algorithm: ED25519

 ED25519 Public-Key:

 pub:

```
        0a:76:42:f1:fa:55:57:a0:c2:b9:de:79:d3:0c:4a:
        50:b8:07:bf:78:ee:0f:45:99:a5:ec:cd:98:07:50:
        c1:33
```

 Attributes:

 (none)

 Requested Extensions:

 Signature Algorithm: ED25519

 Signature Value:

```
    51:83:3a:91:0e:8a:a9:76:58:44:ae:b0:de:a4:3c:7b:50:bd:
    64:0c:6f:ff:ad:6e:6f:f3:97:ea:3e:19:f2:02:f3:0f:b7:8a:
    1c:c1:69:db:b3:8f:35:9e:6d:e8:4a:2f:c8:7a:ce:f2:df:6f:
    3e:e6:80:87:27:7f:1a:64:c4:01
```

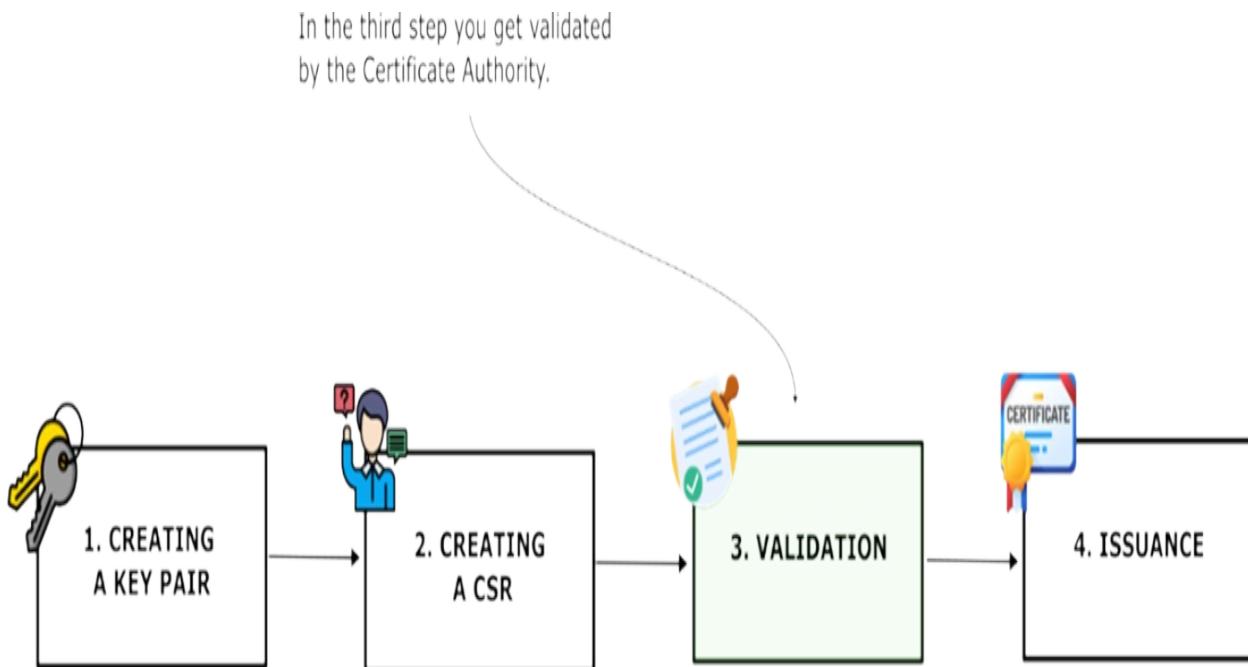
The CSR's Subject field includes the details you entered during the certificate request creation process. You send the CSR to the certificate authority (CA) using a web interface, a proprietary API, email, or an industry-standard protocol like the Automated Certificate Management Environment (ACME) protocol. The CA uses these details to validate your request and issue the certificate.

9.1.3 CSR Validation

Upon receipt of a Certificate Signing Request (CSR), the Certificate Authority (CA) must determine if the request is legitimate (figure 9.4). A

hacker can make a CSR and send it to a CA. The CA must verify the identity of the entity that created the CSR. If the CA can establish the identity of the certificate requester, it can decide whether to issue the certificate or not. What are the ways that CA determines the identity of the CSR requester?

Figure 9.4 Step 3: Time for a background check. In the validation step, the Certificate Authority (CA) takes your CSR and makes sure you are who you say you are. This might involve checking your domain ownership or verifying your organization's identity.



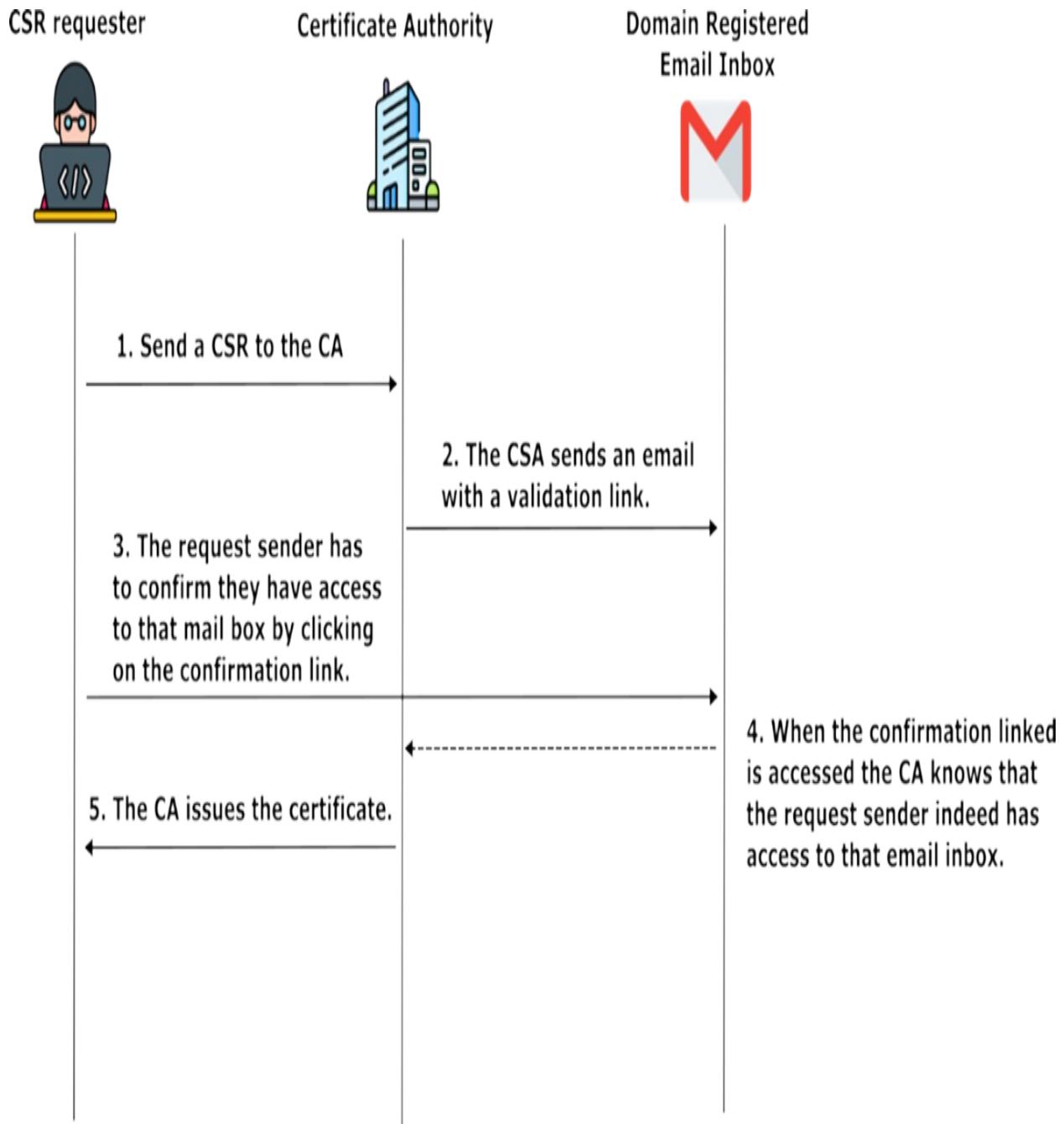
The CSR we created is for a certificate for the example.com domain. The certificate authority needs to validate that the sender of the CSR is, in fact, the owner of the example.com domain. This type of validation is called domain validation because the CA checks that requester of the CSR controls the domain that the CSR is about. There are three common ways to perform domain validation:

- Email - The CA sends a verification email to a registered domain contact.
- HTTP-01 challenge - The CA verifies a specific file hosted on the domain's web server.
- DNS-01 challenge - The CA checks a specific DNS TXT record added to the domain.

In the email challenge the CA sends an email with a validation to the administrator of the domain listed with the domain registrar. If the request is legitimate, the domain administrator clicks on the validation link and follows the CA's instructions to complete the verification process.

The key assumption is that, if someone has access to the *official admin email of the domain*, then they are authorized to ask for certificates for the domain to be issued. This validation process is like the email verification process you have encountered when creating an account on an online service such as facebook.com, or X (figure 9.5).

Figure 9.5 Domain validation via email. This diagram shows how a Certificate Authority (CA) verifies that the CSR requester controls the domain. After receiving the Certificate Signing Request (CSR), the CA sends a validation email. The requester must click a link to prove access to the domain's registered inbox. Once verified, the CA issues the certificate.



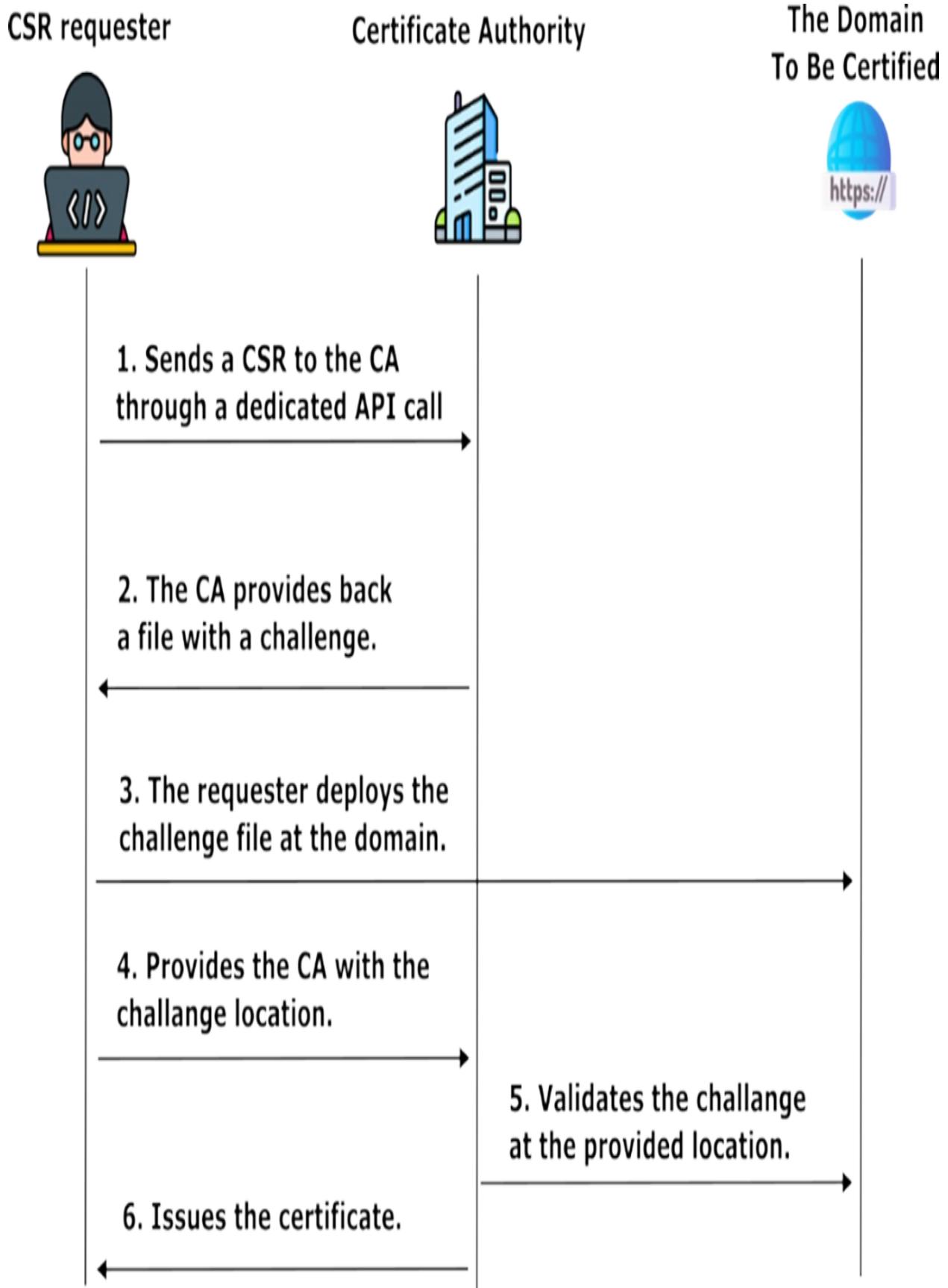
A primary drawback of email-based domain validation is that it is hard to automate. When the verification is complete, the CA issues a certificate and sends the certificate by email to the requester or via a web interface.

HTTP-01 is an industry standard automated way to perform domain validation. HTTP-01 is defined in the ACME protocol in RFC 8555 (<https://datatracker.ietf.org/doc/html/rfc8555>). In this model consists of the

following steps (figure 9.6):

1. Requester makes an API call to the CA over HTTPS to deliver the CSR
2. The CA responds with a small file and a challenge to make the file accessible via HTTP GET on a specific URL on the website of the requestor
3. The requester deploys the challenge file to their website at the specified challenge URL
4. The requester makes an API call to the CA to inform it that it has deployed the challenge file at the challenge URL
5. The CA makes an HTTP GET request to the challenge URL from multiple locations worldwide.
6. If the location contains the challenge file, the CA concludes that the requestor controls the domain's website, and it is ok to issue a certificate. If the CA cannot use HTTP GET to obtain the challenge file, it concludes that the requester does not control the domain, and thus, no certificate should be issued.

Figure 9.6 Domain validation via HTTP challenge. This flow illustrates how a Certificate Authority (CA) validates domain ownership using an HTTP challenge. The requester sends a CSR, receives a challenge file from the CA, and places it at a specific location on their website. When the CA finds the file at that location, it confirms domain control and issues the certificate.

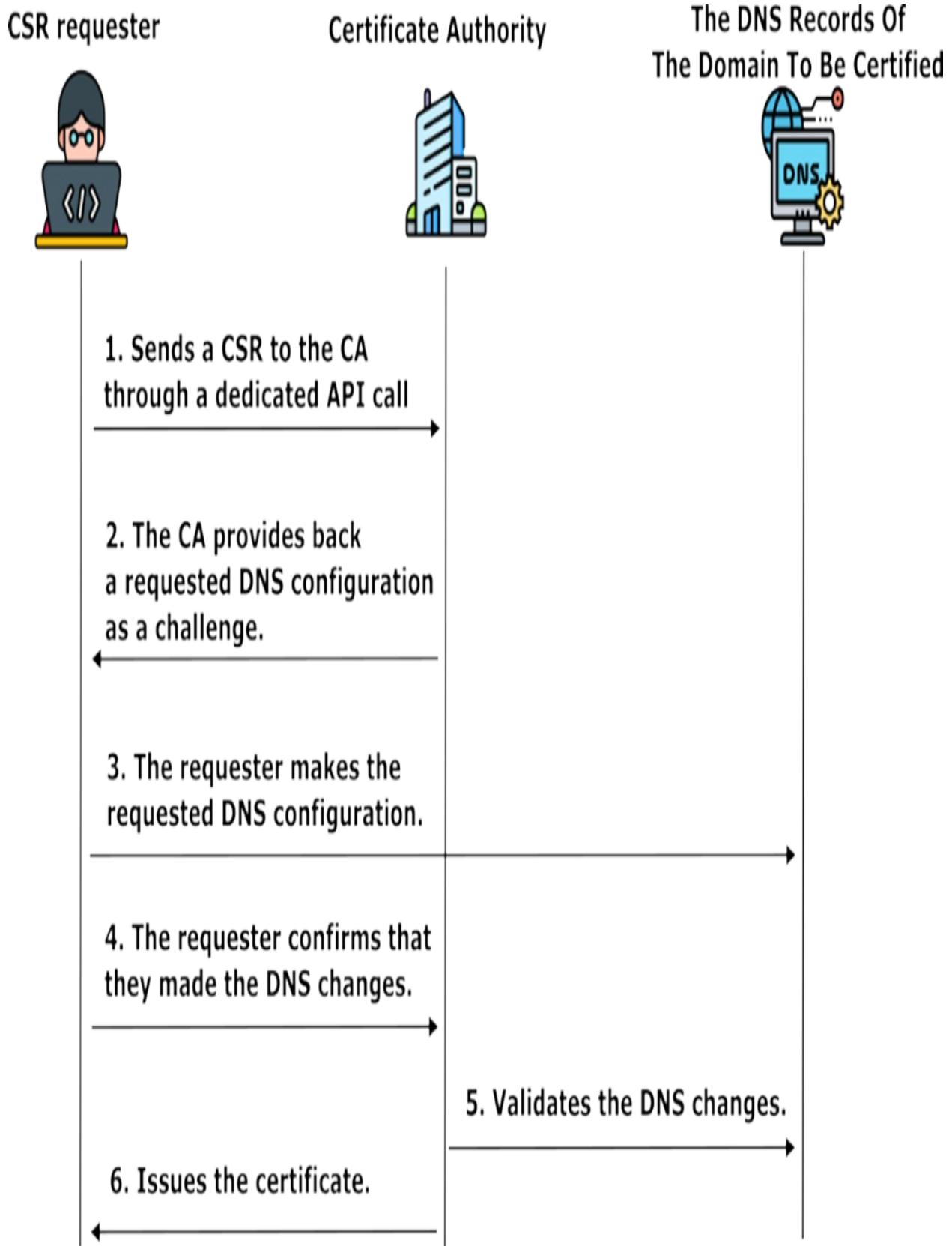


The primary advantage of the HTTP-01 domain validation challenge is that it is an industry-standard with prebuilt automation that makes it possible to request short-lived certificates and automatically renew them, which is a security best practice.

Similar to the HTTP-01 challenge, The ACME protocol defines the DNS-01 domain validation challenge. This challenge consists of the following steps (figure 9.7):

1. Requester makes an API call to the CA over HTTPS to deliver the CSR
2. The CA responds with a challenge that the requester should add a specific record to the DNS configuration of the domain.
3. The requester performs the updates to the domain's DNS configuration
4. The requester makes an API call to the CA to inform it that the requested DNS configuration change is complete
5. The CA checks the DNS configuration of the domain.
6. If the CA finds the DNS record it is looking for, it concludes that the requester has control over the domain since they were able to modify the DNS configuration of the domain, and it is safe to issue a certificate.

Figure 9.7 Domain validation via DNS challenge. This process shows how a Certificate Authority (CA) verifies domain ownership by requiring specific DNS changes. The requester receives a challenge from the CA and updates the DNS records accordingly. Once the changes are live, the CA checks the DNS to confirm control and issues the certificate.



The primary advantage of the DNS-01 challenge is that it does not require the requester to have a website that is accessible from the CA, it can work in situations where HTTP-01 challenge fails.

In summary, the creator of a CSR proves that they can own a certificate for a domain by solving a challenge issued by the CA. The solution of the challenge proves that the requestor has control over the domain in the CSR. There are three common challenge types:

- *Email*: prove control over a domain, by demonstrating the ability to receive emails on the administrative email address listed in the domain registration.
- *HTTP-01*: prove control over a domain my making a challenge file available over HTTP get at a URL chosen by the certificate authority
- *DNS-01*: prove control over a domain by making changes to the DNS configuration of the domain.

HTTP-01 and DNS-01 are fully automated approaches; for example, in Kubernetes, you can use the cert-manager package to implement the HTTP-01 and DNS-01 solvers. We will cover how to use cert-manager in the next section.

TIP

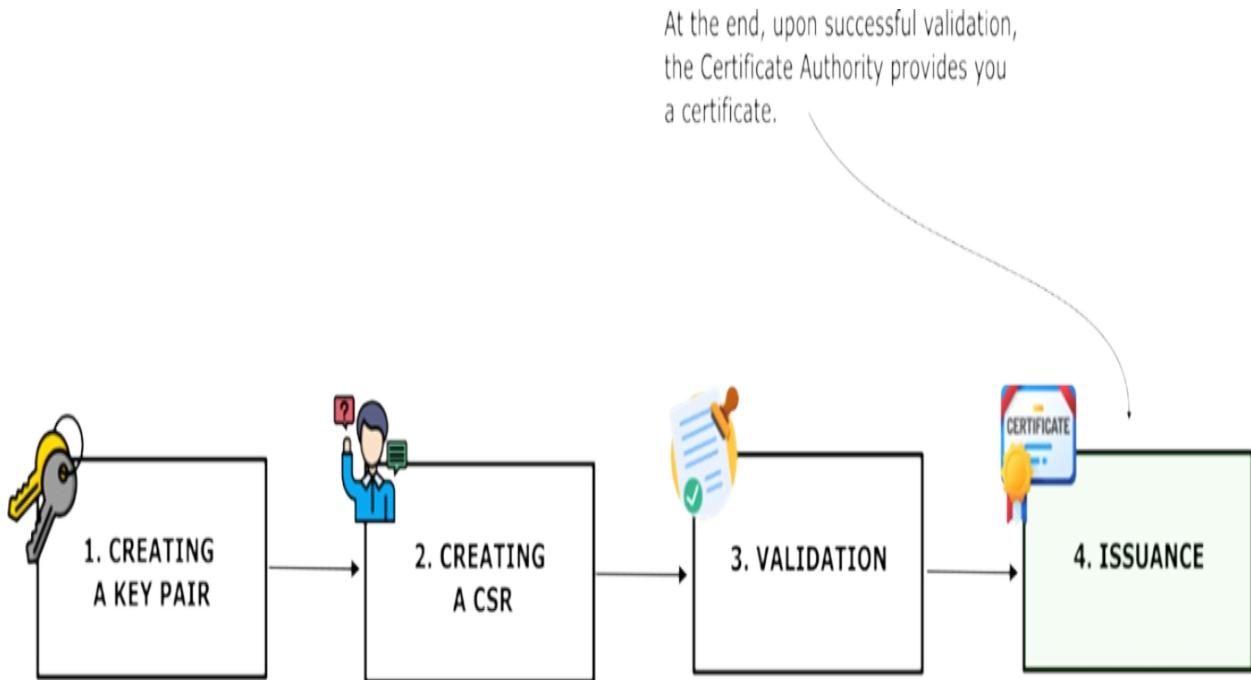
Most certificate authorities charge a fee to issue a certificate. The cost can be from a few dollars to thousands of dollars depending on the certificate authority and the type of certificate. Operating a public certificate authority is complex. It requires dedicated staff to manage and costs millions of dollars annually, so certificate authorities charge fees to cover their operating costs. Let's Encrypt (<https://letsencrypt.org/>) is a non-profit certificate authority that provides certificates at \$0 cost. It was created in 2012 to increase the adoption of TLS on the internet. Let's Encrypt performs domain validation using the HTTP-01 and DNS-01 challenges.

9.1.4 Certificate Issuance

Once the certificate authority has validated that a CSR was created by a

legitimate entity that controls the domain of the requested certificate, the CA creates the certificate and signs it with its private key (figure 9.8). Clients can validate the certificate's authenticity using the rules described in chapter 8.

Figure 9.8 Mission accomplished: certificate issued! After your identity checks out, the Certificate Authority gives you the golden ticket: your digital certificate. This cert is proof that you're trustworthy (at least digitally), and you can now use it to secure communications and earn some internet street cred.



But what if you want to create a certificate without going through a public certificate authority? For example, a checkout microservice calls an inventory microservice over HTTPS. Since both microservices are part of the same application, inside the same company, it might make sense to save the cost of obtaining certificates from a public certificate authority. Instead, you can build an internal certificate authority or use a self-signed certificate.

DEFINITION

A self-signed certificate is one where the subject and issuer of the certificate are the same. You can use the `openssl` to create a self-signed certificate.

The next snippet shows you how to create a self-signed certificate.

```
$ openssl x509 -req -days 365 -in request.csr -signkey private_ke  
➥-out certificate.pem
```

In response to this command you'll get a response similar to what next snippet presents.

```
Certificate request self-signature ok  
subject=C = CA, ST = ON, L = Toronto, O = Example Inc.,  
➥CN = example.com, emailAddress = admin@example.com
```

The command creates an X509 certificate from the request.csr file by signing the generated certificate with the private key that was generated in section 9.1.3. Self-signed certificates are easy to create, but they are not part of the chain of trust. If you use a browser, to access a website secured with a self-signed certificate you will see a scary warning page from the browser. Command line tools like curl, kubectl, and others have options to turn off certificate validation, but this defeats the point of using certificates.

TIP

Don't use a self-signed certificate instead create a private certificate authority and add it to the trust store of your systems; then, your private CA will issue certificates that can be used inside your organization. In section 9.2, we will learn how to create our own certificate authority.

9.1.5 Certificate Revocation

Recall that every certificate has a private key that goes with the public key stored in the certificate. What happens if the public key is accidentally published on the internet, or the private key is stolen before the certificate expires?

In such situations, the certificate owner asks the certificate authority to revoke the certificate. The certificate authority then puts the certificate on its certificate revocation list. During the certificate validation process, the browser and other clients check that the certificate has not been revoked before they decide to trust the certificate.

Certificate revocation is a concept that you should understand as a developer, but it is nothing you need to know how to perform. Typically, DevOps specialists or platform engineers take care of certificate revocation.

9.1.6 Exercises

1. What's the first step in creating a certificate?
2. What's a Certificate Signing Request (CSR)?
3. Why does the CA need to validate the CSR?
4. How does a CA check domain ownership?
5. What is certificate revocation, and why is it important?

9.2 Private Certificate Authority for Local Development

Most developers use HTTP during local development and don't try developing with TLS and HTTPS, which can lead to if statements in application logic and scripts to account for the on-laptop vs. on-cluster deployment of the application. When TLS is used, self-signed certificates are used. Still, self-signed certificates generate many errors when they are consumed, which leads to more command line options passed to tools to ignore certificate validation errors, or even worse, it leads to application code that turns off certificate validation, which gets into production and can lead to security breaches.

TIP

Use TLS for local development on your laptop. Do it by creating a personal certificate authority for your laptop. Install the certificate authority into your laptop, then create as many certificates as you need, and you should end up with no certificate validation errors without turning off certificate validation. Creating a personal certificate authority for local development is not difficult. Now, let me show you how.

9.2.1 Create a Self-Signed Root Certificate

Let's create a self-signed certificate. We'll have to follow a few steps. The first step is to use `openssl` to generate a keypair to use as the private key of the certificate authority. The next snippet shows you how to generate the key pair.

```
openssl genpkey -algorithm RSA -outform PEM -out ca/ca_private_ke
```

This command outputs the key pair into the file at the path `ca/ca_private_key.pem`, which we can turn into an authority certificate that is valid for a given period of time using the `openssl` command. The following snippet shows the command that creates an authority certificate valid for 10 years.

```
openssl req -x509 -new -key ca/ca_private_key.pem  
➥-days 3650 -out ca/ca_cert.pem -subj "/CN=local-dev CA"
```

We can inspect the generated certificate authority certificate using the command in the next snippet to understand exactly what the certificate contains and to verify that it was created correctly. This inspection helps confirm important details such as the subject (who the certificate is for), the issuer (who signed it should be the same as the subject for a self-signed CA), the validity period (when it starts and when it expires), and the public key associated with it.

Additionally, we can check the certificate extensions to ensure it includes the necessary flags for a certificate authority, such as Basic Constraints: CA:TRUE and appropriate key usages. All this information gives us confidence that the CA certificate is valid and configured as expected before it is used to sign other certificates.

```
openssl x509 -in ca/ca_cert.pem -noout -text
```

Using this command we can read the certificate details. Listing 9.4 gives you these details as they have been printed in the console as a result of the previous given command.

Listing 9.4 The details of the generated certificate

Certificate:

Data:

 Version: 3 (0x2)

 Serial Number:

 0d:0f:db:21:94:67:c0:cb:cd:10:cf:58:ba:d8:c8:5f:49:58

 Signature Algorithm: sha256WithRSAEncryption

 Issuer: CN=local-dev CA

 Validity

 Not Before: Jun 21 15:30:50 2024 GMT

 Not After : Jun 19 15:30:50 2034 GMT

 Subject: CN=local-dev CA

 Subject Public Key Info:

 Public Key Algorithm: rsaEncryption

 Public-Key: (2048 bit)

 Modulus:

 00:ba:fa:3e:66:1f:a2:12:e2:ca:75:cb:23:55:0e:
 39:46:b5:61:de:2b:2d:46:56:0c:b9:cc:9c:b7:87:
 5d:c3:2b:4a:58:f2:e2:1e:07:01:d6:f6:a6:5d:77:
 bd:84:5b:1d:9c:c9:ea:49:5b:36:0e:2f:75:ea:0c:
 68:fa:e2:c8:a3:c7:9a:a3:cb:5f:cd:f1:bb:5c:b3:
 41:8d:34:81:d8:53:38:5b:aa:95:85:44:3b:d0:9d:
 93:58:06:a8:29:5a:00:91:4a:ab:a0:7c:31:30:fc:
 cb:1d:76:60:28:71:e0:a0:8f:fd:cf:08:d3:29:1a:
 f9:1a:af:2c:bb:06:01:50:a8:d7:86:6d:2f:ff:25:
 2f:29:91:bf:18:ce:63:5e:32:a7:a9:d1:17:1a:0e:
 b9:3b:54:fe:40:2d:36:75:b2:03:f4:05:51:24:36:
 51:d1:74:60:77:48:a7:a4:b3:72:f2:2d:32:0e:7c:
 84:04:a9:a3:9f:0c:d3:55:f4:14:17:30:b2:1c:e6:
 e2:82:18:29:a4:27:d2:23:0e:70:54:30:60:79:ca:
 c5:dc:cc:18:46:b3:b6:48:3a:b8:19:71:3c:00:f4:
 b9:93:bd:15:96:b8:34:25:8d:1a:6d:16:ee:a4:80:
 d4:40:93:10:6a:45:d4:2e:d0:61:08:f8:c9:c7:c6:
 33:3b

 Exponent: 65537 (0x10001)

X509v3 extensions:

 X509v3 Subject Key Identifier:

 B2:93:61:82:15:EC:9C:BA:AD:17:EF:C2:48:87:22:FE:8

 X509v3 Authority Key Identifier:

 B2:93:61:82:15:EC:9C:BA:AD:17:EF:C2:48:87:22:FE:8

X509v3 Basic Constraints: critical

CA:TRUE #A

 Signature Algorithm: sha256WithRSAEncryption

 Signature Value:

 1e:98:1d:ba:02:92:03:28:73:da:db:79:a1:ee:9f:09:e2:50:

 56:14:0b:f6:2d:09:86:12:94:0a:ec:16:cd:fd:a8:10:73:a8:

 b4:d7:7c:e6:6a:18:e7:fd:d9:13:fc:04:f1:02:d4:fb:d9:e0:

 3d:79:8c:dc:c9:64:e1:17:2a:1b:3f:4c:82:cd:e2:1b:10:dd:

 3d:4b:d4:6e:c6:f5:be:0d:e4:79:c0:fc:e2:ac:b2:35:52:c7:

```
8d:0d:43:50:f6:e6:dc:76:5e:b0:fc:32:69:c3:a0:a1:2f:0d:  
2a:79:c3:58:15:ec:fa:73:8e:bc:4d:4b:7a:28:d0:d1:c2:30:  
40:b8:90:c2:4e:f7:89:72:ec:c2:95:ba:60:94:e4:29:ad:61:  
01:13:e0:d1:1c:f2:94:ae:ee:1b:fe:76:b1:43:b2:be:1a:01:  
fb:da:34:07:5e:c9:e8:c2:e1:41:18:d6:71:b5:72:ff:64:e1:  
f9:df:66:ba:5a:aa:ff:bc:ee:17:f5:b9:67:e0:28:48:58:3f:  
21:e1:8a:7c:10:c2:eb:2d:77:a2:7c:a4:28:97:03:31:20:c1:  
79:f6:d9:e7:6c:92:46:76:aa:6b:4a:3e:f8:ad:b8:66:67:74:  
7b:9a:4f:94:50:3e:8c:eb:30:49:13:ea:84:12:95:7d:5f:5b:  
4e:80:24:12
```

9.2.2 Install the Certificate Authority into the Operating System Trust Store

We have created a certificate authority (CA) certificate, but it won't be trusted by anyone yet. As explained in chapter 8, operating systems come with a pre-installed list of trusted certificate authorities provided by the OS vendor. To make our CA certificate trusted, we need to add it to the operating system's list of trusted certificate authorities.

Once added to the system's trusted list of authorities, any certificate issued by our CA will pass validation. Keep in mind that the process for installing the CA certificate varies depending on the operating system, so you'll need to follow specific instructions for macOS, Windows, or Linux.

On MacOS we can install the certificate authority using the command presented in the next snippet.

```
sudo security add-trusted-cert -d -r trustRoot  
➥ -k "/Library/Keychains/System.keychain" ca/ca_cert.pem
```

On Linux systems, the steps to add a certificate authority (CA) certificate differ between distributions. On Ubuntu, you need to copy the CA certificate file to the directory where Ubuntu stores CA certificates and then run a command to reload the trusted certificates (see an example in the following snippet).

```
sudo cp ca/ca_cert.pem /usr/local/share/ca-certificates/myCA.crt  
sudo update-ca-certificates
```

On Windows machines you will need to use the windows “Microsoft Management Console” this is a GUI application, check the Microsoft documentation at (<https://learn.microsoft.com/en-us/windows-hardware/drivers/install/trusted-root-certificationAuthorities-certificate-store>) for the details on how to do the certificate install.

Installing a local development certificate authority is specific to each operating system, so the instructions provided may become outdated over time. The critical concept is that every operating system maintains a list of trusted certificate authorities.

By adding our development CA to this list, the operating system will trust any certificates we issue. A quick web search usually provides up-to-date, step-by-step instructions for installing CA certificates. Once you understand the key concepts, the process is straightforward to follow.

9.2.3 Issue a certificate using the personal certificate authority

Now that we have certificate authority for our development machine, we can issue a server certificate that we can use in our application, such as adding it to a Spring Boot application to turn on TLS. We start by creating a key pair for the server certificate as shown in the next snippet.

```
openssl genpkey -algorithm RSA -outform PEM -out server_key.pem
```

The certificate must have a Subject Alternative Name defined. Otherwise, we will get an error from the browser. We must create an OpenSSL configuration file to pass as an input parameter in the command.

Listing 9.5 OpenSSL Configuration file ca/server-cert.cnf

```
[ req ]  
distinguished_name = req_distinguished_name  
req_extensions     = req_ext  
x509_extensions   = v3_ca  
prompt             = no  
  
[ req_distinguished_name ]  
C                   = Canada
```

```

ST          = Ontario
L           = Toronto
O           = Adib Saikali
OU          = MacBook Pro
CN          = localhost

[ req_ext ]
subjectAltName = @alt_names

[ v3_ca ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1    = localhost

```

Using the configuration file we can command to create a certificate signing request based on the inputs in the configuration file.

```
openssl req -new -key server_key.pem -out server_csr.pem
➥-config ca/server-cert.cnf
```

Next, we sign CSR using the certificate authority certificate we created earlier.

```
openssl x509 -req -in server_csr.pem -CA ca/ca_cert.pem
➥-CAkey ca/ca_private_key.pem -CAcreateserial
➥-out server_cert.pem -days 365 -sha256 -extensions req_ext
➥-extfile ca/openssl.cnf
```

We now have a certificate that we created with our certificate authority. We can use OpenSSL to check if the certificate is valid using the command

```
openssl verify server_cert.pem
```

This command produces the following output:

```
server_cert.pem: OK
Chain:
depth=0: C=CA, ST=Ontario, L=Toronto, O=Adib Saikali,
➥OU=MacBook Pro, CN=localhost (untrusted)
depth=1: CN=local-dev CA
```

We now have a server certificate that we can use in our applications. In chapter 10, we are going to use the certificate and the certificate authoring to explore the TLS. Give yourself a pat on the back, you have learned a lot about certificates and you now have the background to do something real with them.

9.2.4 Exercises

6. Why would you create a local certificate authority (CA)?
7. What's a self-signed certificate?
8. Why not just use self-signed certificates for everything?
9. How can you make your local CA trusted by your OS?
10. What's the benefit of using your CA instead of skipping TLS locally?

9.3 Answers to exercises

1. What's the first step in creating a certificate?

You start by creating a key pair: a private key (you keep this secret) and a public key (you can share). This is the foundation of every certificate.

2. What's a Certificate Signing Request (CSR)?

It's like a digital application form. You send your public key and some personal info to a Certificate Authority (CA) asking for a certificate.

3. Why does the CA need to validate the CSR?

So the CA knows you're not a random person trying to get a certificate for something you don't own, like someone else's website.

4. How does a CA check domain ownership?

It can send you a special email, ask you to put a file on your website (HTTP challenge), or update your DNS settings (DNS challenge). Each proves you control the domain.

5. What is certificate revocation, and why is it important?

Revocation means telling everyone "Don't trust this cert anymore", maybe because the private key got stolen or the cert is no longer needed. It's a way to undo trust.

6. Why would you create a local certificate authority (CA)?

So you can issue your own certificates for development without paying

or waiting on a public CA. Great for testing, bad for public use.

7. What's a self-signed certificate?

It's a certificate you sign yourself instead of having a public CA sign it.

It works for testing, but browsers won't trust it by default.

8. Why not just use self-signed certificates for everything?

Because they aren't part of the global chain of trust. Most systems and browsers will throw scary warnings or just reject them.

9. How can you make your local CA trusted by your OS?

You install the CA certificate into your operating system's list of trusted certificate authorities. The method depends on whether you use macOS, Linux, or Windows.

10. What's the benefit of using your CA instead of skipping TLS locally?

You can test real-world HTTPS setups, avoid writing "if TLS" hacks in your code, and keep your security behavior consistent from dev to prod.

9.4 Summary

- The certificate lifecycle includes critical steps such as generating a key pair, creating a certificate signing request (CSR), validating the request, issuing the certificate, and revoking it when necessary.
- OpenSSL enables the creation of a key pair consisting of a private key and its corresponding public key, which form the foundation of an X.509 certificate.
- A Certificate Signing Request (CSR) includes the public key and metadata about the certificate, which is sent to a Certificate Authority (CA) to request certificate issuance.
- Domain control is verified using one of three methods: email-based validation, hosting a challenge file on the website (HTTP-01), or adding a specific DNS TXT record (DNS-01).
- The Certificate Authority (CA) validates the CSR, ensures the requester controls the domain, and issues a signed certificate that can be used for secure communication.
- Self-signed certificates allow independent creation of certificates without a public CA, but they are not trusted by browsers or systems without additional configuration.
- A local Certificate Authority (CA) can be established for development environments, allowing developers to issue trusted certificates within

their own systems.

- Adding a local CA certificate to the operating system's trusted certificate list ensures that all certificates issued by the local CA are recognized and validated by the system.
- Developers can use their private CA to issue certificates for internal applications, avoiding the need to rely on external Certificate Authorities for testing purposes.

10 Transport Layer Security (TLS): How the internet is secured

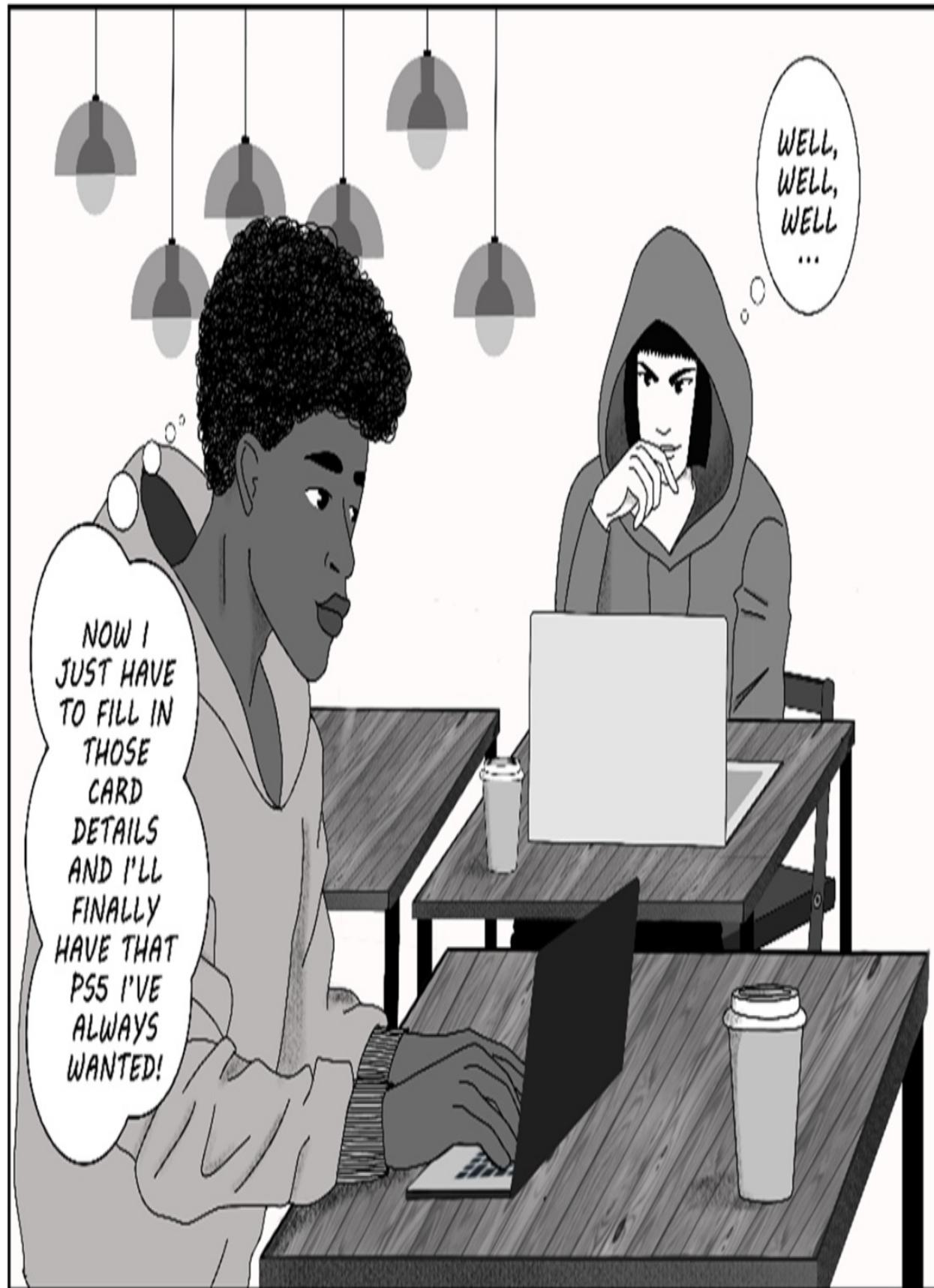
This chapter covers

- Understanding how TLS secures communication over untrusted networks
- Discovering what TLS actually protects you against, and what it doesn't
- Applying TLS works in practice across browsers and servers
- Identifying common pitfalls in TLS configuration and how to avoid them

The internet is like a noisy city full of people who love to listen in. Every time your phone or computer sends something, like logging into your bank, ordering sushi, or sending a file, it's a bit like whispering a secret in a crowded Starbucks. If you're not careful, anyone nearby can hear it. Even worse, someone might grab the message, change your tiramisu to a cheesecake, or pretend to be your bank.

TLS gives you an encrypted briefcase (so no one can read your message), a signature wax seal (so they know it's really from you), and a way to confirm that the recipient isn't some villain in disguise. It's the technology behind that little padlock in your browser. And without it, the web would be a free-for-all of stolen passwords, fake websites, and identity theft (figure 10.1).

Figure 10.1 When surfing the Internet from a public network, you never know who shares the network with you. For this reason, all communications need to be secured.



We'll start in section 10.1 by looking at how TLS builds trust in a hostile network, where anyone could be listening or tampering with your data. From there, in section 10.2, we'll break down what TLS actually protects you against, including eavesdropping, data tampering, and impersonation. In section 10.3, we'll get practical: you'll see how TLS is used in real applications, how it's configured on servers, and what can go wrong if it's set up poorly.

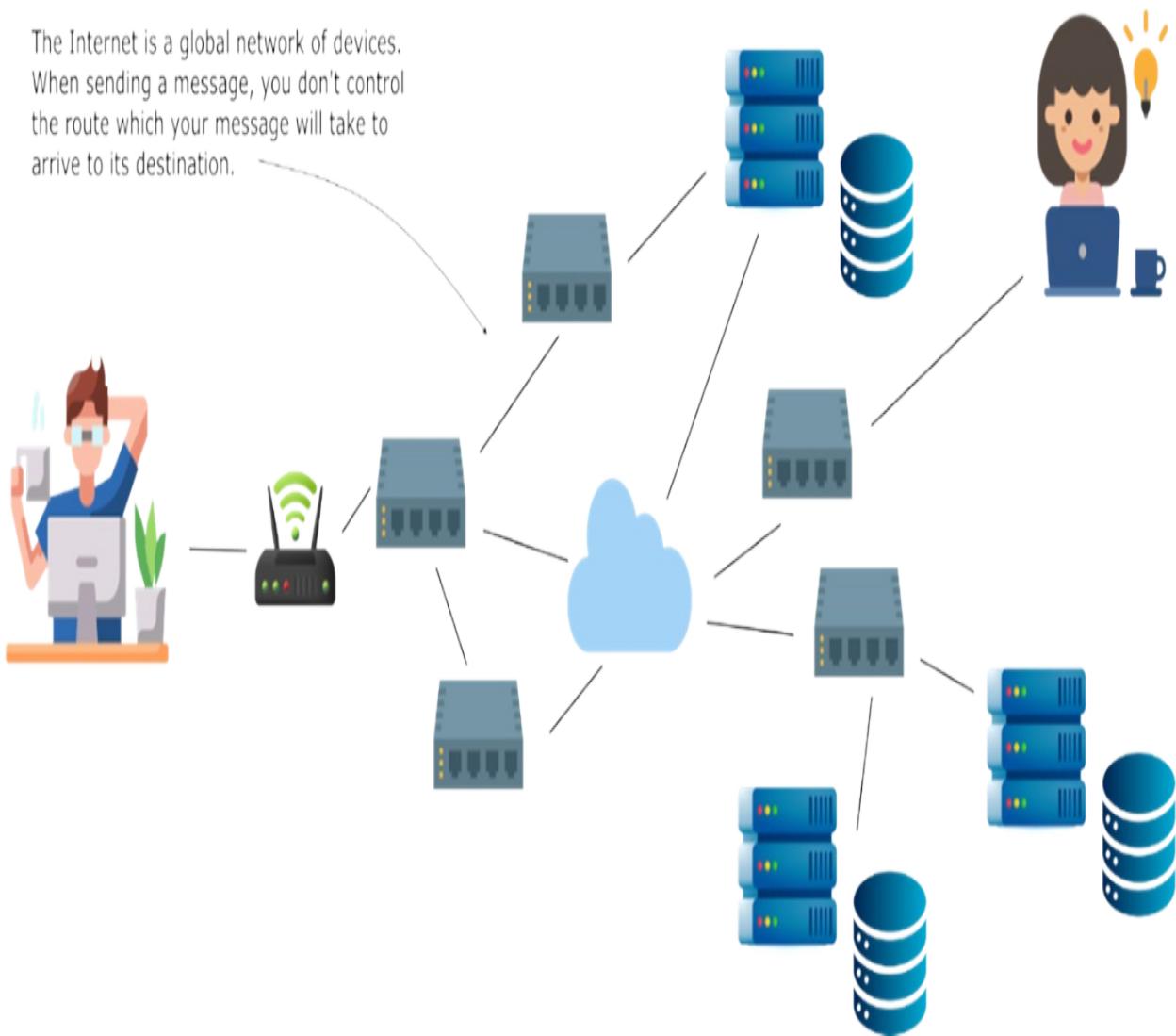
By the end of the chapter, you'll clearly understand how TLS works and why it's a critical foundation for securing the modern internet.

10.1 Securing communication with TLS

The internet wasn't built with trust in mind. It was built to send messages from point A to point B, even if that meant crossing through point C, D, and some random router in Iceland. Along the way, your data can pass through dozens of machines you don't control and don't even know they exist (figure 10.2).

Figure 10.2 The internet is a massive, unpredictable network. When you send a message, it travels through many unknown devices and routes before reaching its destination, which is why encryption, like TLS, is essential.

The Internet is a global network of devices. When sending a message, you don't control the route which your message will take to arrive to its destination.



That's the problem: on the internet, you don't get to choose who carries your messages. You have to assume the network is hostile, because sometimes it is. Hackers, spyware, compromised Wi-Fi networks, or misconfigured routers can become uninvited middlemen.

So how do you build trust in a place like that?

That's where TLS comes in. It doesn't try to make the network safe. It works despite the network being unsafe. It gives your app a way to talk in code, prove who it's talking to, and be sure the message wasn't changed along the way. In this section, we'll explore why this kind of trust is so hard to get, and how TLS manages to pull it off.

10.1.1 How it all started

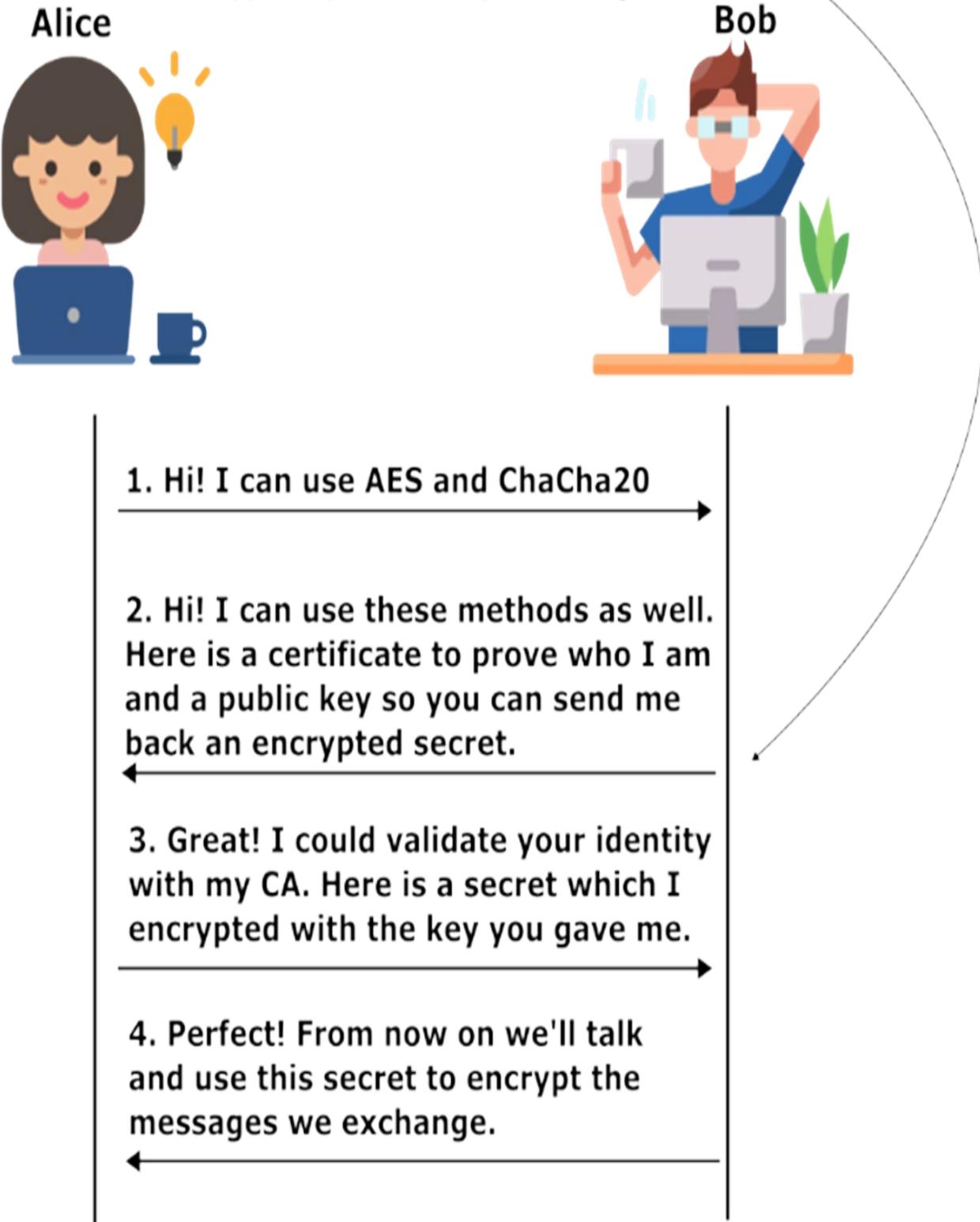
If you are old enough, you might have heard about *Secure Socket Layer* (SSL). Before TLS became the quiet guardian of secure internet connections, there was SSL. Think of SSL as the chunky but brave ancestor of TLS: it had the right idea, but its armor had a few holes.

SSL is like this (figure 10.3):

1. Alice (your browser) says “Hi!” and tells Bob (the server) what types of secret codes (encryption methods) she understands.
2. Bob replies with his own list and sends Alice a certificate: a digital ID card that proves who he is and gives her his public key.
3. Alice checks this ID (using a certificate authority she trusts) and uses Bob’s public key to send back a shared secret.
4. From that point on, both use the same secret key to encrypt and decrypt their messages.

Figure 10.3 The SSL handshake in action: Alice and Bob exchange supported encryption methods, validate identities using certificates, and securely agree on a shared secret used to encrypt all future communication.

With SSL, the two parties will exchange a secret which they use further to encrypt any data they exchange.



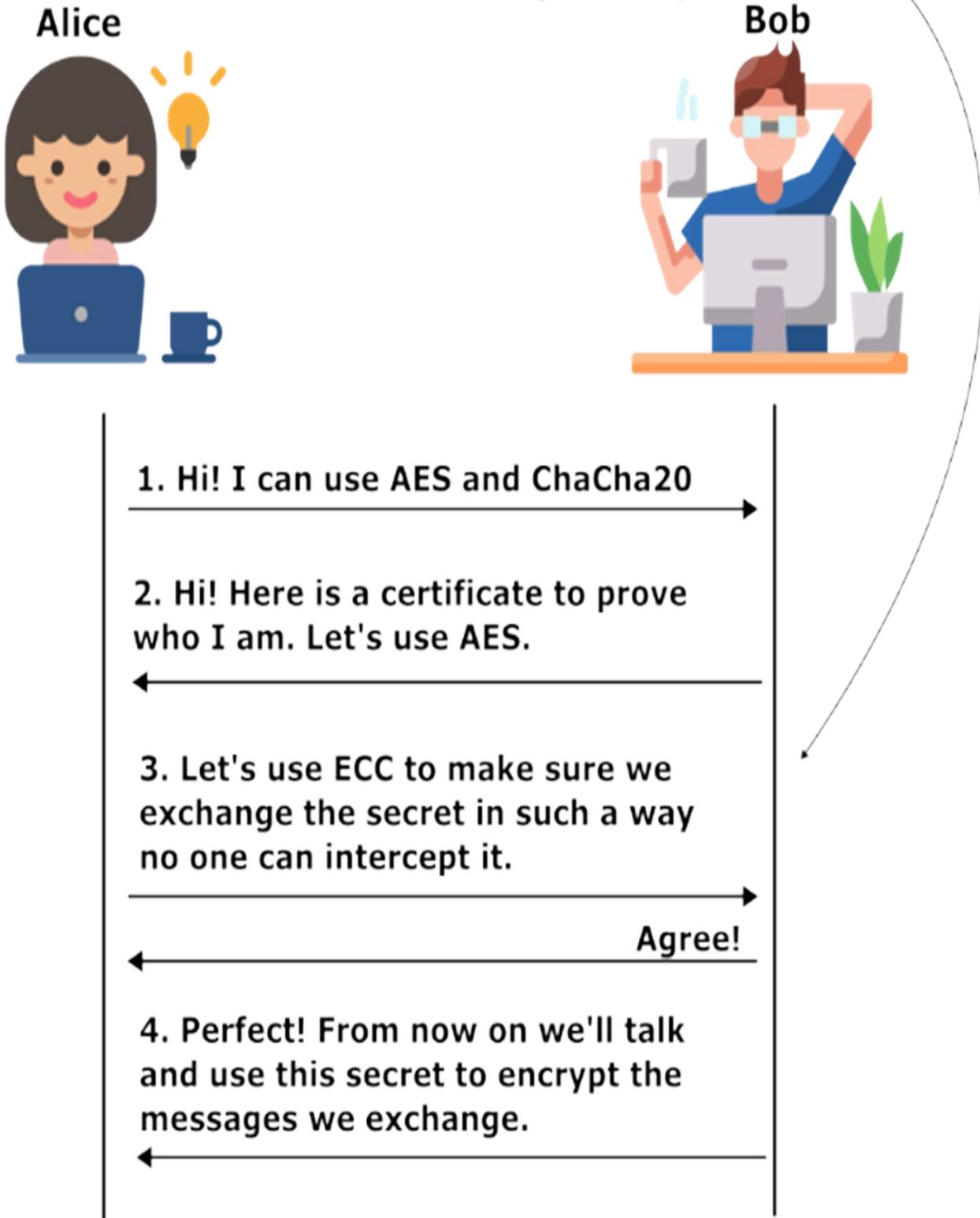
This sounds good enough, but hackers found issues. They found a way to trick browsers into using weaker encryption (this is called a downgrade attack). They have also found that sometimes they could steal parts of the session key by analyzing patterns, and sometimes they simply exploited code bugs.

TLS works a bit differently, and each step has been upgraded to be a lot more robust than SSL (figure 10.4):

1. Alice (your browser) wants to talk securely with Bob (a website).
Hi Bob! I'd like to talk in private. Here are the encryption methods I support.
2. Bob replies with his digital certificate.
Hi Alice! Here's my digital certificate to prove I'm really Bob. Let's use this encryption method.
3. They use some clever math (like elliptic curve cryptography) to agree on a shared secret, without actually sending the secret over the network.
4. Once they both have the secret, they use it to encrypt everything they say from now on.

Figure 10.4 In TLS, Alice and Bob agree on encryption methods, exchange certificates, and use elliptic curve cryptography (ECC) to securely generate a shared secret without ever sending that secret across the network.

With TLS, the two parties use ECC to exchange a secret they can then use to enhance messages safely



Now I know what you are thinking. Point 3 sounds like magic isn't it? Once Alice and Bob agreed on an encryption method, how do they actually use that "clever math"? Well, this clever math is indeed not simple to explain, but let me try an analogy to give you a top of the mountain picture on how this works.

Imagine Alice and Bob's secret is actually a color. How can they decide on a color that no one else knows even if they are listening to what they talk to each other? To start, they both agree on a common color: yellow. This color is public, and everyone can see it.

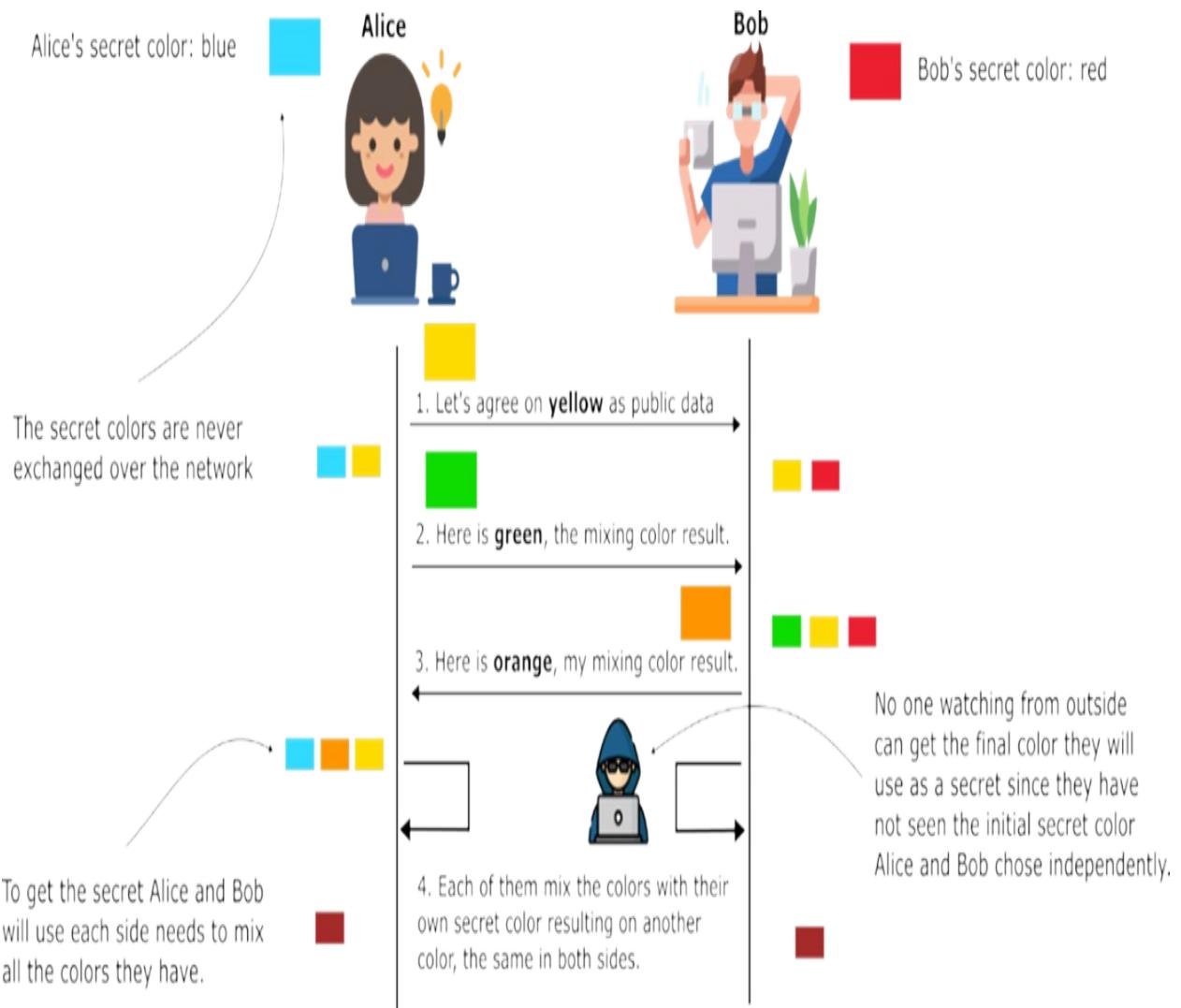
Alice now picks her own secret color: blue. She mixes her blue with the public yellow and gets green. She sends this green mixture to Bob.

Meanwhile, Bob picks his secret color: red. He mixes it with the same public yellow and ends up with orange. He sends the orange mix to Alice.

Alice takes Bob's orange mix (yellow + red) and adds her secret blue. Bob takes Alice's green mix (yellow + blue) and adds his secret red. Even though they're mixing things independently, they both end up with the same final color.

Anyone watching this exchange only saw the yellow, green, and orange. They can't recreate the final shared color without knowing Alice's blue or Bob's red. That's how Alice and Bob agree on a secret, without ever revealing it (figure 10.5).

Figure 10.5 Using color mixing as an analogy for key exchange in TLS: Alice and Bob each choose a private color, mix it with a public one, and exchange the results. They then mix again locally to arrive at the same final color without ever revealing their secret. An outsider, even if watching the entire exchange, can't reconstruct the final shared secret.



In TLS, the same idea from the paint analogy is used. Except that instead of colors, Alice and Bob use numbers. These numbers are combined using special math functions. They're designed to be easy to compute in one direction (mixing), but nearly impossible to reverse (unmixing). One of the most common methods today is elliptic curve cryptography, which is both fast and secure. Remember we talked about this in chapters 6 and 7.

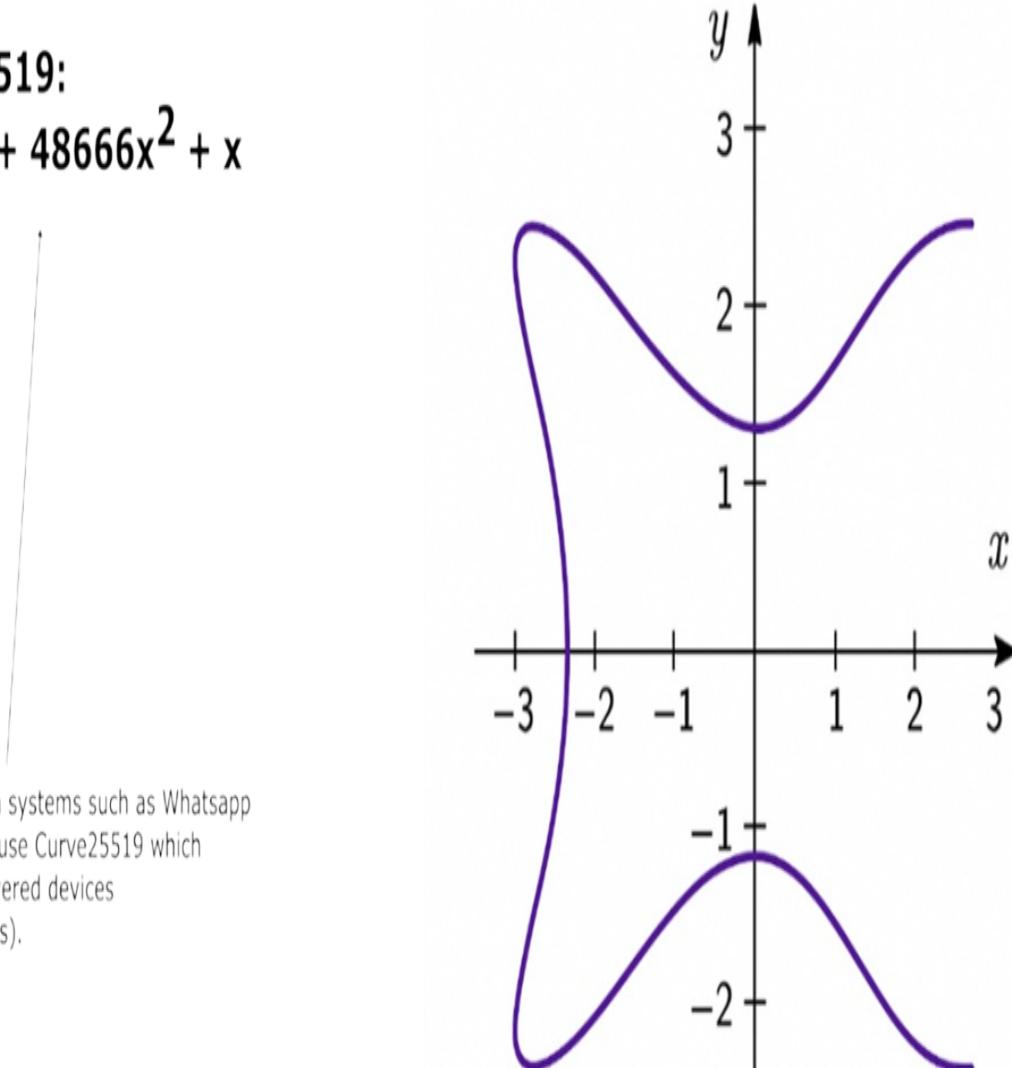
At the heart of ECC is a special type of math function called an elliptic curve. It's not a stretched-out oval like the name suggests. It's actually a curve defined by this equation:

$$y^2 = x^3 + ax + b$$

This equation draws a smooth, looping curve on a graph with interesting properties. As an example and to be able to visualize such a curve, figure 10.6 shows you an example of a curve named Curve25519. It is usually used by systems running on low-powered devices.

Figure 10.6 Curve25519: A widely used elliptic curve in secure communication systems like WhatsApp and Signal. Its equation allows both high security and efficient performance, making it ideal for modern encryption on low-powered devices like smartphones.

Curve25519:
 $y^2 = x^3 + 48666x^2 + x$



Secure communication systems such as WhatsApp and Signal commonly use Curve25519 which is efficient for low-powered devices (such as smart phones).

What makes it useful for cryptography is that you can take two points on this curve, "add" them together, and get another point on the curve. This

"addition" isn't like normal math, it's geometric. You draw a line through two points, and where it intersects the curve again, you reflect that point over the x-axis. That's the result.

If you keep adding the same point to itself over and over, (like doubling, tripling, etc.), you predictably move along the curve. But if someone only sees the final result, it's very hard to figure out how many times you added the point to get there. This is called the Elliptic Curve Discrete Logarithm Problem, and it's what makes ECC secure.

ECC turns simple curve math into a one-way trapdoor. A function that is easy to go forward, but nearly impossible to reverse without the key. That's why it's used in TLS, secure messaging, cryptocurrencies, and more.

Even if someone on the network sees all the messages being exchanged, just like someone watching the color mixing, they still can't figure out the final shared secret. That's because the private parts (like Alice's blue or Bob's red) are never sent. Only the mixed results are visible.

The beauty of this approach is that the shared secret never travels across the network. Each side calculates it independently, using the public information and their own private key. This makes the communication secure, even if the entire exchange happens over an insecure or hostile network.

TLS isn't just a better version of SSL. TLS is the modern standard. In fact, all versions of SSL are now considered unsafe and have been officially retired. TLS picked up where SSL left off, fixing its flaws and strengthening every part of the handshake. Since 2018, TLS 1.3 has become the preferred version.

10.1.2 TLS and mTLS

TLS, or Transport Layer Security, is the technology that puts the "S" in "HTTPS." It's what makes sure that when your browser connects to a website, no one can read or change the information being exchanged. It does three main things: it encrypts the data so others can't see it, it checks that the data wasn't tampered with, and it verifies that the server you're talking to is the real one, not an impostor.

This is how most internet connections work: your browser checks the server's certificate, proves it's talking to the right website, and then sends data securely. But notice something: only the server proves who it is. Your browser doesn't need to prove anything about itself. This works fine for many public-facing websites where the user doesn't need to be authenticated through the connection itself (they'll just log in with a password later).

But this one-sided trust isn't enough in more secure environments, especially in backend systems where services talk to other services. That's where mutual TLS (mTLS) comes in.

With mTLS, both the client and the server present certificates, and both verify each other. It's like a secret handshake where each side must prove their identity before the conversation starts. This ensures that not only is the server trusted, but so is the client. It's extremely useful in microservice architectures, APIs exposed to trusted partners, or any system where you want to block unknown or rogue clients from even establishing a connection.

Here's how mTLS works (figure 10.7):

1. Alice (a client service) wants to talk securely with Bob (a server service).
Hi Bob! I'd like to talk in private. Here are the encryption methods I support. Also, here's my digital certificate to prove who I am.
2. Bob replies with his own certificate.
Hi Alice! Thanks for your certificate. Here's mine to prove I'm really Bob. Let's use a strong encryption method.

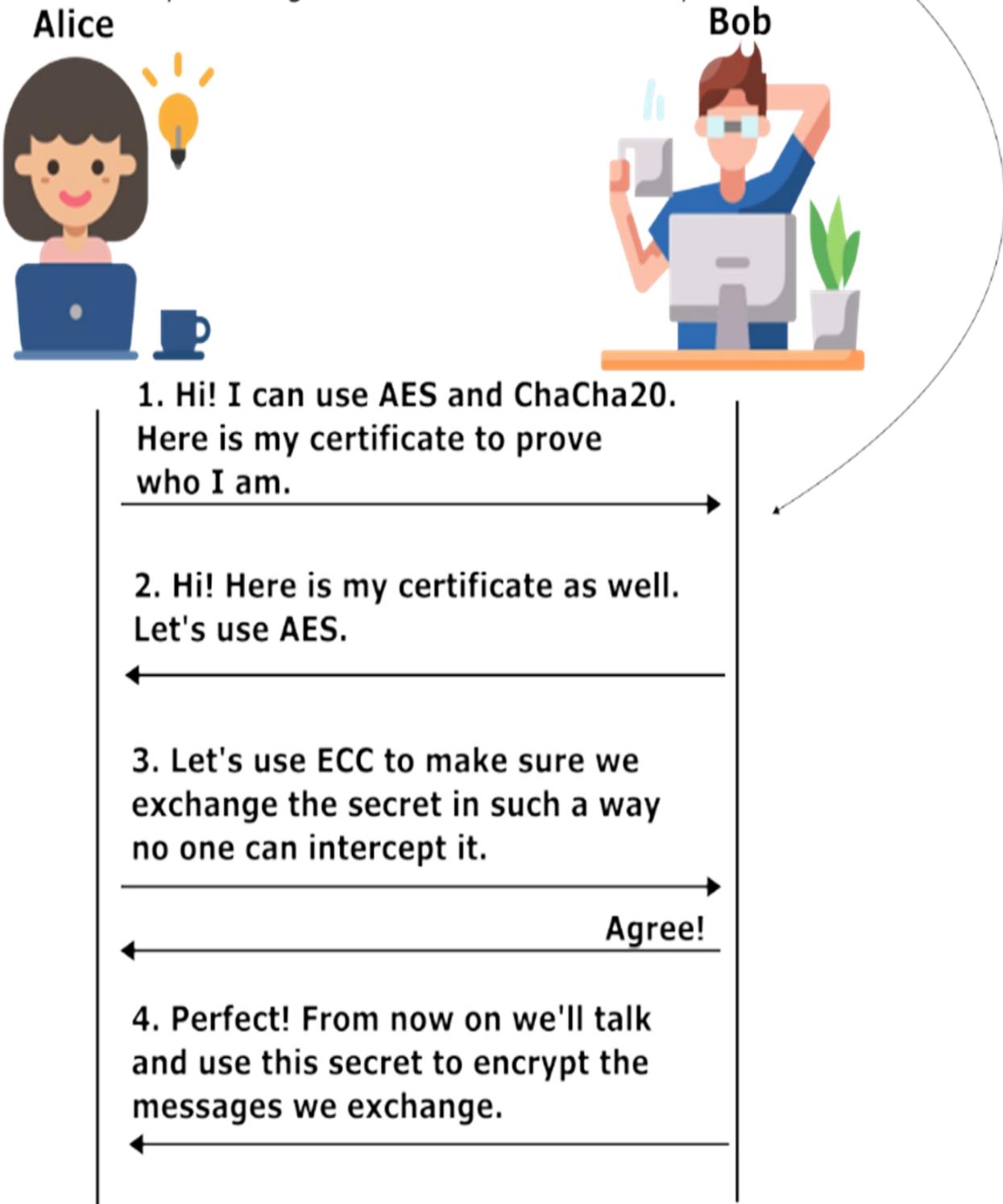
Both check each other's certificates, making sure they're valid and trusted.

Alice checks that Bob's certificate was signed by someone she trusts. Bob does the same for Alice's certificate.

1. They use some clever math (like elliptic curve cryptography) to agree on a shared secret, without actually sending it across the network.
2. Once they both have the shared secret, they use it to encrypt everything they say from that point on.

Figure 10.7 In mutual TLS (mTLS), both Alice and Bob present valid certificates to prove their identities before establishing a secure connection. Once identities are verified, they use elliptic curve cryptography (ECC) to safely agree on a shared secret for encrypting their communication.

In mTLS both participants in the conversation need to prove their identity with a valid certificate issued by a recognized certificate authority.



Setting up mTLS is more complex than regular TLS because now both the client and the server need certificates. In standard TLS, only the server has to prove who it is. You get a certificate for the server (usually from a certificate authority like Let's Encrypt - <https://letsencrypt.org/>), install it, and you're good to go. But with mTLS, every client also needs its own certificate, a digital ID that proves who it is when connecting.

That means you must create, distribute, and store client certificates safely. You also need to track which certificates are trusted, which are expired, and what to do when a client's certificate is compromised. In practice, this involves running your own certificate authority (CA).

It is extra work, but it gives you a big security boost: you can control which clients can connect to your services. If a device or service doesn't present a valid certificate, the connection is immediately rejected, even before any data is exchanged. This is much stronger than relying on just usernames, passwords, or API keys, which can be leaked or guessed.

So while mTLS adds complexity, it also gives you fine-grained, cryptographic control over who can talk to whom in your system. That's why it's often used in internal microservices, or financial and healthcare environments where trust must be explicit and verified at every connection.

We'll dive deeper into these service-to-service communication patterns, including best practices for using mTLS, in chapter 16. Table 10.1 summarizes our discussion about TLS and mTLS.

Table 10.1 A comparison between TLS and mTLS

	TLS	mTLS
Encrypts data	Yes	Yes
Server proves identity	Yes	Yes
Client proves identity	No	Yes
Common use	Web browsing	Service-to-service communication and secure APIs

Client cert required	No	Yes
----------------------	----	-----

10.1.3 Exercises

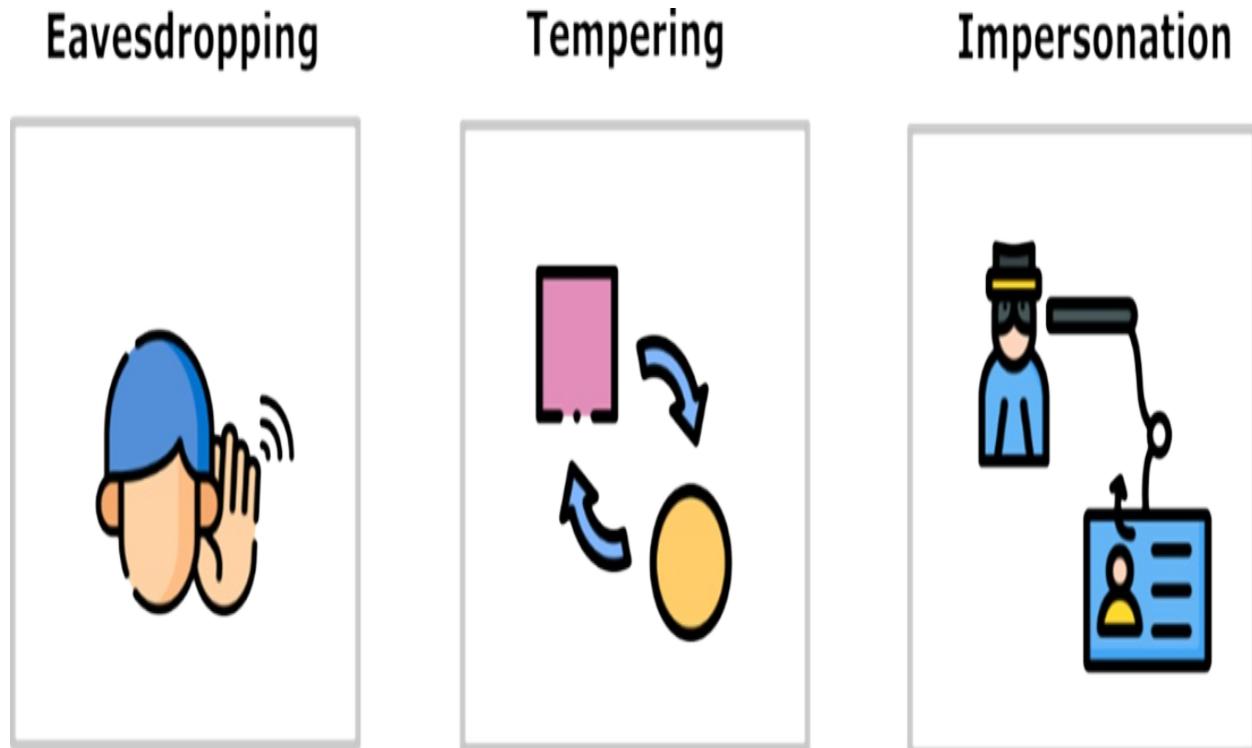
1. Why do we need TLS on the internet?
2. What is the TLS handshake?

10.2 What TLS actually protects you against

When browsing a website, sending an email, or using an app, your device is tossing sensitive information into a network you don't control. TLS acts like a shield, protecting that data in three key ways (figure 10.8):

- Eavesdropping (Keeping Things Private)
- Tempering (Making Sure Nothing Changes)
- Impersonation (Knowing Who You're Talking To)

Figure 10.8 TLS protects against three major threats: eavesdropping (unauthorized listening), tampering (modifying data in transit), and impersonation (pretending to be someone else using fake credentials).



10.2.1 Protection from eavesdropping

Without TLS, anyone sitting between you and the server can read your messages. That includes Wi-Fi snoopers in a café, compromised routers, or malicious internet service providers. TLS solves this by encrypting everything you send or receive. Even if someone intercepts the data, all they see is nonsense.

This isn't just theory. On an unsecured public Wi-Fi network (like in an airport, hotel, or café), someone using freely available tools can "sniff" the traffic and see exactly what websites people are visiting, what forms they're filling out, or what files they're downloading. The same thing can happen if an attacker has access to a router or DNS server somewhere along the path. Even worse, attackers can modify what you receive, injecting fake pages, pop-ups, or malware.

TLS protects against all of that by encrypting the connection from start to finish. It works like sealing your message in a locked, tamper-proof envelope that only the real recipient can open. Once TLS is in place, even if someone manages to intercept the traffic, they see a stream of random-looking encrypted data. They don't know what you sent, and they can't change it without being detected.

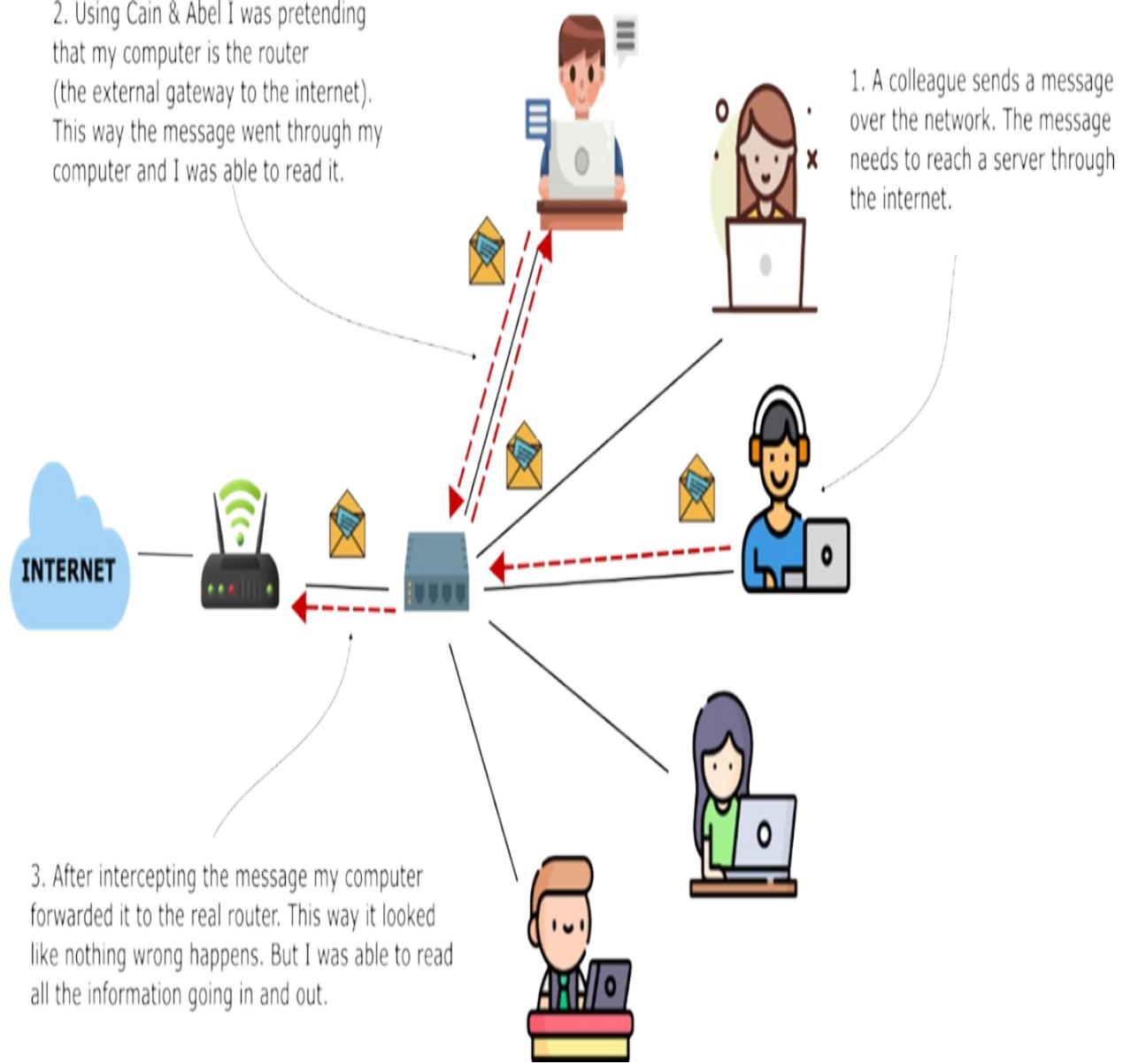
Back in high school, I used to run a tool called Cain & Abel on the school network. It was a network sniffer, a piece of software that let me intercept and read the messages my classmates were sending to each other over Yahoo Messenger and other chat apps during class. And yes, it worked. I could see the full conversation, plain as day. Why? Because most of those apps didn't use encryption at the time. No TLS, no SSL. Just raw, readable text floating through the network.

Cain & Abel actually tricked the network into trusting my computer as if it were the main router. It did this by faking the MAC address (a kind of hardware ID) of the real router. As a result, all the internet traffic in the classroom's local network, such as messages, websites, and everything, got rerouted through my machine before going out to the internet (figure 10.9).

Figure 10.9 A man-in-the-middle attack using Cain & Abel: By impersonating the router, an

attacker silently intercepts messages between users and the internet. The traffic is forwarded normally to avoid suspicion, but the attacker can read all data in and out.

2. Using Cain & Abel I was pretending that my computer is the router (the external gateway to the internet). This way the message went through my computer and I was able to read it.



This is what's called a *man-in-the-middle attack*. The other students' devices thought they were talking directly to the internet, but in reality, they were talking to me first, and I could read everything before passing it along (including passwords). It was like standing in front of a mailbox, opening each letter, and then resealing it and sending it on its way.

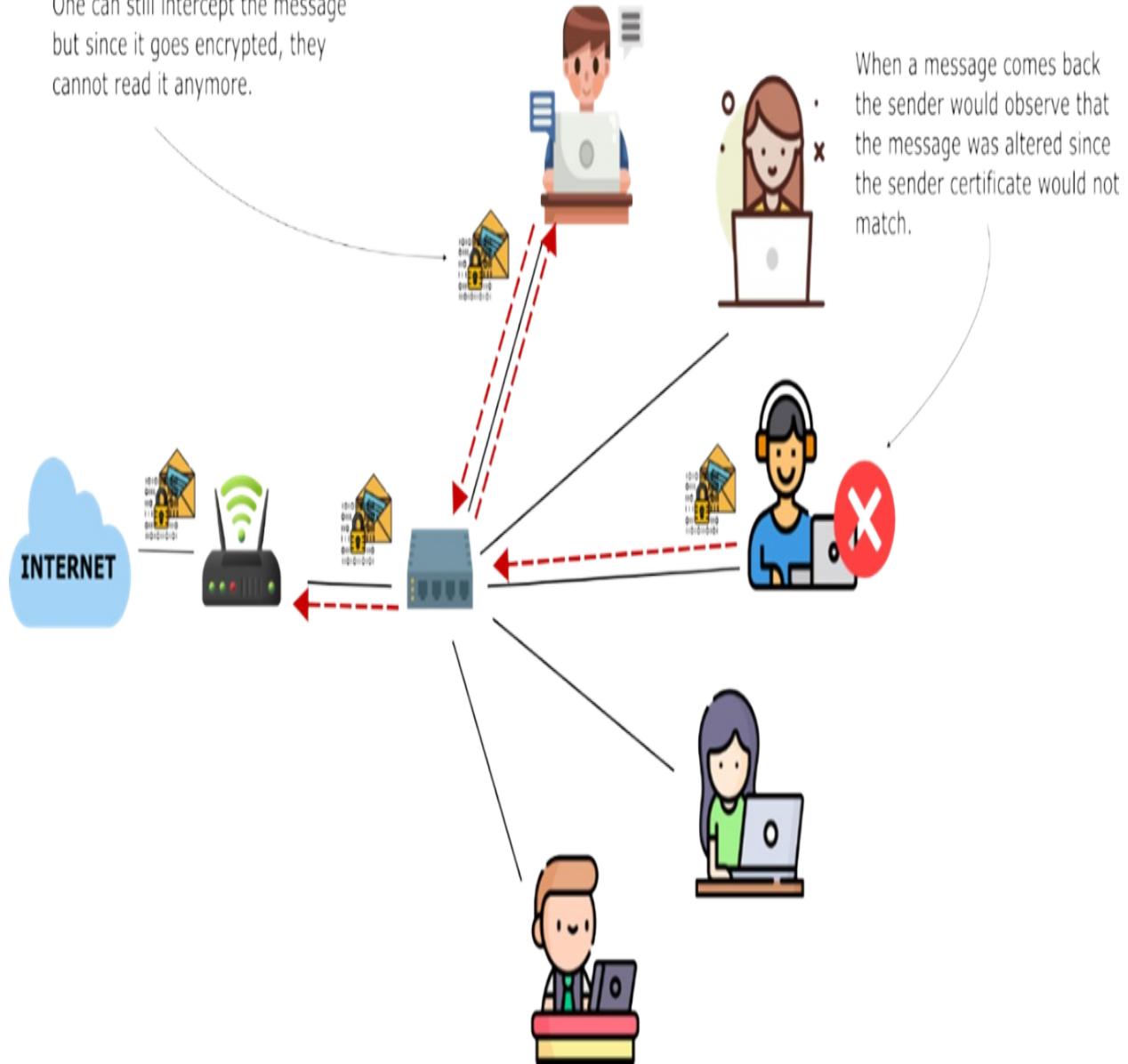
And it worked because most services didn't use encryption back then. Without TLS or SSL, the messages were just plain text. I didn't need to hack

anything. I just intercepted what was already out in the open.

Today, that kind of attack would be much harder to pull off, because TLS is everywhere (figure 10.10). It encrypts the connection between your device and the server so that even if someone intercepts the data, all they get is encrypted gibberish. TLS turns open messages into sealed boxes that only the right recipient can unlock. Without it, you're basically shouting secrets in a crowded room.

Figure 10.10 With TLS enabled, intercepted messages are encrypted and unreadable to attackers. Even if someone tries to alter the message, the recipient will detect the tampering because the digital signature won't match the sender's certificate. This was TLS ensures both confidentiality and integrity.

One can still intercept the message but since it goes encrypted, they cannot read it anymore.



10.2.2 Making sure there's no tempering

Even if no one reads your data, what if they silently change it? Imagine you're downloading a file, maybe a software update or a bank statement, and someone in the middle swaps it out with something else, like a virus or a fake version. Or think about filling out a payment form online. You enter the correct bank account number, but a hidden attacker quietly replaces it with their own. You hit “Send,” and your money goes to the wrong person.

Note

Tempering is usually harder to notice than eavesdropping because everything still looks normal to the user.

TLS protects against this with something called message integrity. Every piece of data you send or receive is digitally signed using something like a cryptographic checksum (also called a MAC or an HMAC). We discussed message integrity more in detail in chapter 4. You can think of it like a tamper-evident seal on a package. If someone opens the box or changes what's inside, the seal breaks, and TLS will know that something isn't right.

When your browser sees that the message doesn't match the expected integrity check, it immediately discards it. So TLS doesn't just keep your data private—it also makes sure that what you receive is exactly what the sender intended, no matter how many routers, servers, or unknown devices it passes through.

10.2.3 Avoiding impersonation

What if the server you're talking to isn't really your bank, but a clever fake that looks exactly like it? Same logo, same login page, same everything. You could type in your password and credit card number without realizing you're sending them straight to an attacker. This is called spoofing or server impersonation, and it's a real threat, especially on open Wi-Fi networks or in phishing attacks.

TLS solves this problem using digital certificates. When your browser connects to a secure site (one that starts with https://), the server sends its certificate as part of the TLS handshake. That certificate includes important information: who the website claims to be, when the certificate expires, and, most importantly, a digital signature from a trusted certificate authority (CA).

Your browser already knows and trusts a list of these certificate authorities. So when it sees a certificate from a website, it doesn't just take the site's word for it. It checks that the certificate is:

- Valid and not expired
- Properly signed by a trusted CA
- Issued for the right hostname (e.g. bank.com, not báñk.com – observe

the small difference in letter “a”)

If anything looks off (e.g. expired, forged, mismatched name), your browser throws up a big red warning or blocks the connection entirely. This is what protects you from accidentally talking to a fake website that’s trying to steal your data (figure 10.11).

Figure 10.11 Browsers indicate a secure TLS connection with a padlock icon next to the URL. Clicking the icon reveals certificate details, such as the issuing Certificate Authority, the site’s public key, and verification that your connection is encrypted and trustworthy.

Always make sure you are protected by TLS. In your browser, if you are navigating using HTTPS a small lock will appear next to the address bar.



Click on the small certificate icon to get more details about the certificate, the public key the browser uses, and the CA that issued the certificate.

You can even click the padlock icon in your browser's address bar to view the certificate details. Most people never do, but it's one of the most important pieces of online security happening quietly behind the scenes.

10.2.4 Exercises

3. What are the three main threats TLS defends against?
4. What is a man-in-the-middle (MITM) attack?
5. How does TLS detect tampering?
6. How do browsers know they're talking to the real website?
7. What happens if a certificate is expired or fake?

10.3 TLS in practice

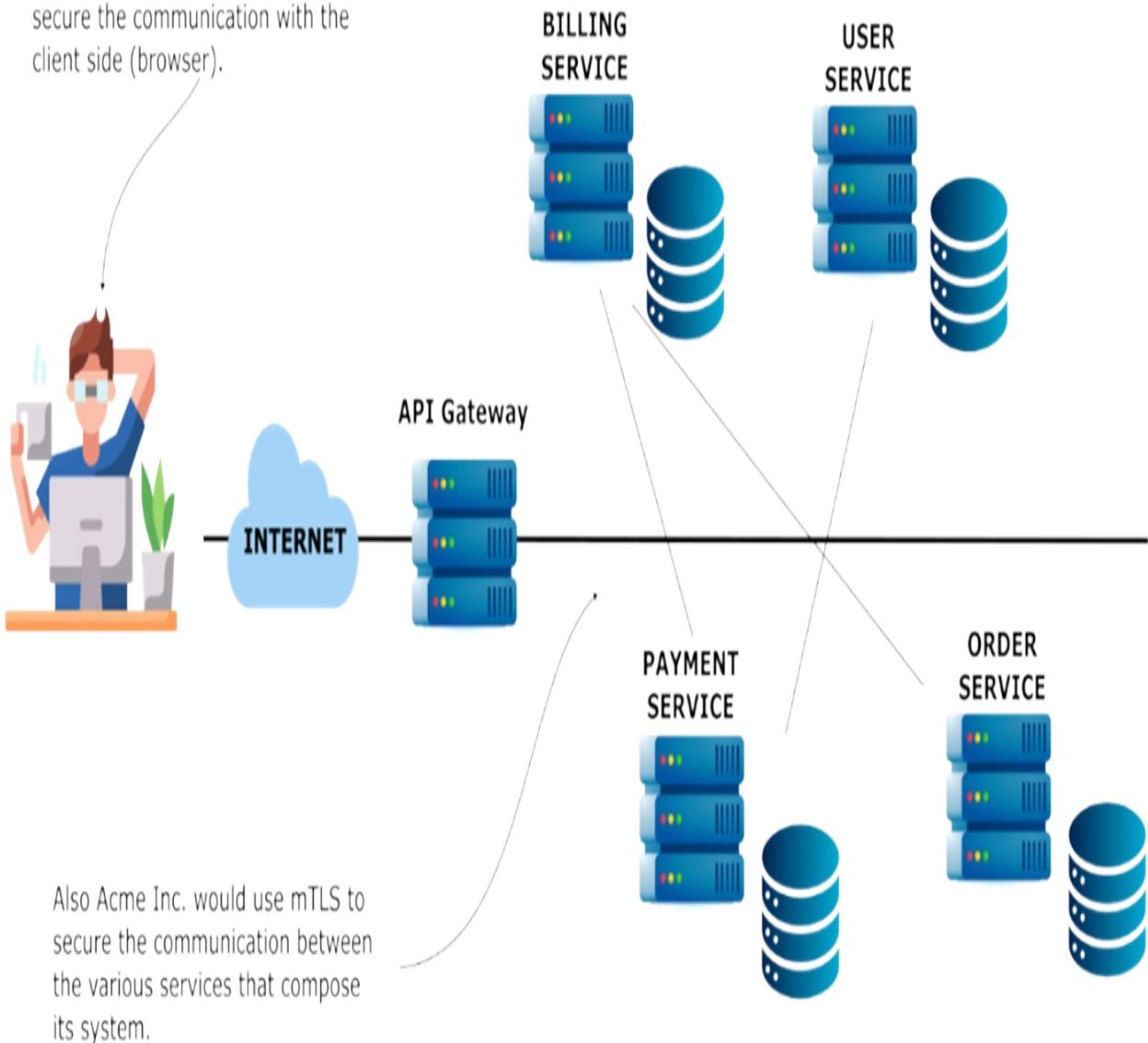
At this point, you understand what TLS is and why it matters. But now it's time to bring it down to earth, into the real systems you're building and running daily. How should you apply TLS to our Acme Inc. to enhance the system's security? In this section, we'll look at:

- How to enable TLS on your public-facing services
- How to secure internal service communication with mTLS
- Real-world configuration examples in Spring Boot and Kubernetes

Let's start with the basics: Acme Inc.'s customer-facing web app, exposed through a domain like `https://app.acme.com`. This app is built with Spring Boot and runs in Kubernetes behind an ingress controller (figure 10.12).

Figure 10.12 Acme Inc. uses TLS to secure communication between users and the system via an API Gateway. For internal service-to-service communication, such as between billing, payment, order, and user services, mTLS is used to ensure both encryption and mutual authentication within the system.

Acme Inc. system needs to use TLS to secure the communication with the client side (browser).



Your users must trust that they're really talking to acme.com, and their credentials and personal data must be encrypted. That's where TLS comes in.

When users connect to your app over `https://`, their browser starts a secure TLS connection. That means all the data they send, like login details or form submissions, is encrypted. But the question is: where should that encrypted connection end? Where should the data be decrypted so your app can read it?

In Spring Boot, you can terminate TLS directly inside the application. This means your app needs its own TLS certificate and private key, which must be configured to accept secure connections (usually on port 8443). It works, but

it also comes with downsides. You now have to manage certificates for every app individually, ensure they're renewed before expiring, and handle TLS configuration in every project. It also complicates your infrastructure, especially in Kubernetes, because traffic must stay encrypted in the pod.

A much better and more common approach, especially in large systems like Acme Inc.'s, is to terminate TLS at the Ingress level. Ingress is the entry point into your Kubernetes cluster. It's like the front door. When a user connects to <https://app.acme.com>, the Ingress controller (like NGINX or Traefik) decrypts the data there, checks the certificate, and then forwards the unencrypted request to your Spring Boot app inside the cluster.

Definition

Ingress is a service orchestrator (Kubernetes) component that acts as the entry point for external traffic into your cluster. It routes incoming HTTP or HTTPS requests to the right service inside the cluster, and often handles things like TLS termination, load balancing, and URL path routing.

Note

In large systems, it's much more common, and often recommended to terminate TLS at the Ingress level rather than inside each application.

Two of the most common Ingress controllers are NGINX (<https://nginx.org/>) and Traefik (<https://traefik.io/>). NGINX is one of the most well-known web servers in the world. It's reliable, powerful, and has been around for a long time. When used as an Ingress controller, NGINX can handle things like routing requests to the right service, redirecting from HTTP to HTTPS, and even applying rules like rate limits or timeouts. It works really well and is used by many large companies, but it can require more manual configuration.

Traefik, on the other hand, is newer and built with Kubernetes in mind. It's designed to be easy to set up and automatically detects your services and routes without needing a lot of extra configuration. Traefik also has built-in support for automatically requesting and renewing TLS certificates using Let's Encrypt, and it even comes with a nice dashboard to monitor traffic and status.

Both are excellent tools. NGINX is a solid choice if you want fine control and don't mind writing a bit more YAML. If you prefer something more dynamic and easier to manage, Traefik might be a better fit, especially for fast-moving environments like Acme Inc.'s.

A setup using an ingress is simpler and more scalable. You only need to manage certificates in one place. Tools like cert-manager can even renew and install certificates automatically. Your Spring Boot apps don't need to worry about TLS at all. They just run on plain old HTTP, which keeps the app code and configuration clean.

You can think of it like a secure building: the ingress is the front desk with security. Once a visitor gets past the front desk, they can walk through the building. But if you want to protect sensitive rooms deeper inside, you can always add more checks (and that's where mTLS comes in for internal service communication). Listing 10.1 shows a sample of configuration for the ingress with NGINX.

Listing 10.1 Configuring TLS in Kubernetes with NGINX Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: acme-app
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod #A
    nginx.ingress.kubernetes.io/ssl-redirect: "true" #B
spec:
  tls:
    - hosts:
        - app.acme.com #C
      secretName: acme-app-tls #D
  rules:
    - host: app.acme.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: acme-webapp
                port:
```

number: 80

The certificate will be automatically requested by cert-manager, validated via DNS or HTTP challenge, and mounted as a secret named acme-app-tls. NGINX Ingress takes care of serving traffic securely.

As discussed, the recommended way is using an ingress. However, in some particular cases (e.g. admin tools), you may want your Spring Boot app to terminate TLS itself. In such a case, application frameworks such as Spring and its ecosystem make configurations easier. With a Spring Boot app, you'd just have to configure the certificate with a few configuration lines as presented in listing 10.2.

Listing 10.2 Enabling TLS in a Spring Boot app

```
server.port=8443
server.ssl.key-store=classpath:keystore.p12 #A
server.ssl.key-store-password=changeit #B
server.ssl.key-store-type=PKCS12 #C
server.ssl.key-alias=acmeapp
```

Important

In real-world apps, passwords such as the key store password should be stored securely (e.g. using environment variables or external secrets).

Make sure the keystore file contains your private key and certificate (you can generate it with the Java keytool or use a proper CA).

To enable mutual TLS (mTLS) for a Spring Boot app, you need to configure the application to not only present its own certificate but also require and validate a certificate from the client. This involves setting up a truststore (to validate incoming client certificates) alongside the keystore (which holds the app's own identity). It adds a strong layer of authentication between services.

However, in most modern deployments (especially in Kubernetes), mTLS is often handled by a service mesh, like Istio, so the Spring Boot code stays clean. We'll explore mTLS service configuration, including trust stores,

certificate rotation, and service-level authentication, in more detail in chapter 16.

TLS is one of those technologies that does its job quietly. Whether you’re securing a customer-facing app, exposing APIs, or managing dozens of services inside your cluster, getting TLS right is essential. And while tools like Spring Boot, Kubernetes Ingress, and Istio make it easier than ever to set up, how you configure TLS still matters, a forgotten setting, an expired certificate, or a weak cipher suite can turn a locked system into an open one.

At Acme Inc., your systems now have a solid foundation: the front door is locked (thanks to HTTPS and Ingress), and internal doors can be locked too when needed (via mTLS). You’ve seen how to apply these patterns with real code, and more importantly, how to think about TLS in practice.

10.3.1 Exercises

8. Why use ingress controllers for TLS in Kubernetes?
9. What’s the difference between NGINX and Traefik?
10. When would you still do TLS termination inside the app?

10.4 Answers to exercises

1. Why do we need TLS on the internet?

Because your data passes through a bunch of unknown systems, some of which might be spying. TLS encrypts it so no one else can read or tamper with it.

2. What is the TLS handshake?

It’s the initial exchange where two sides agree on encryption methods, share certificates, and safely agree on a shared secret using clever math.

3. What are the three main threats TLS defends against?

Eavesdropping (reading your data), tampering (changing it), and impersonation (pretending to be a trusted site).

4. What is a man-in-the-middle (MITM) attack?

It’s when someone sits between you and the server, silently intercepting or modifying messages. TLS prevents this by encrypting everything.

5. How does TLS detect tampering?

Each message includes a cryptographic signature. If someone changes the data, the signature won't match and the message is rejected.

6. How do browsers know they're talking to the real website?

They check the digital certificate from the server (verifying the issuer, expiration date, and hostname match).

7. What happens if a certificate is expired or fake?

The browser shows a big red warning and may block the connection entirely to protect the user.

8. Why use ingress controllers for TLS in Kubernetes?

They simplify things. You manage certificates in one place instead of each app, and traffic inside the cluster can stay unencrypted or use mTLS.

9. What's the difference between NGINX and Traefik?

NGINX gives more manual control. Traefik is more dynamic and integrates better with Kubernetes and Let's Encrypt.

10. When would you still do TLS termination inside the app?

For standalone apps (like internal tools) or in cases where you want end-to-end encryption all the way into the application.

10.5 Summary

- TLS protects data as it travels across untrusted networks by encrypting it, verifying the identity of the server, and ensuring the data hasn't been changed along the way.
- TLS solves three major problems:
 - eavesdropping (no one else can read the message)
 - tampering (no one can change it without being detected)
 - impersonation (you know you're talking to the real server)
- Before TLS, there was SSL - a now-retired, less secure version. TLS 1.3 is the modern standard, using faster and safer encryption methods like elliptic curve cryptography.
- mTLS (mutual TLS) extends TLS by requiring both sides to prove who they are. It's often used in secure backend systems where services must trust each other explicitly.
- TLS is typically terminated at the Ingress level (e.g., using NGINX or Traefik) in Kubernetes. This simplifies certificate management and keeps service configuration clean.

- In special cases, Spring Boot apps can terminate TLS themselves using a configured keystore, though this approach is more complex to scale.

Part 4: Modern Authentication and Identity

In Part 3, you learned how to establish trust between systems using certificates and TLS. That gave you the foundation for secure communication. But trust at the system level isn't enough. Applications also need to know who the user is and what they're allowed to do. This is where authentication and identity protocols come into play.

Part 4 is all about people and services proving who they are. We'll explore the standards and technologies that power modern authentication — from JSON-based formats like JWS, JWE, and JWT (chapter 11), to OAuth2 and OpenID Connect, which enable Single Sign-On (chapter 12) and advanced identity management (chapter 13). You'll then see how passwordless login options like magic links and one-time passwords (chapter 14) work, before moving on to WebAuthn (chapter 15), which brings hardware tokens, biometrics, and phishing-resistant authentication into the mainstream.

Throughout this part, you'll see how Acme Inc. applies these protocols in real scenarios: securing APIs with tokens, simplifying user logins with an identity provider, and protecting accounts with stronger authentication methods. You'll also learn about the trade-offs — why some flows are safe in theory but risky in practice, and how to implement best practices to avoid common pitfalls.

By the end of Part 4, you'll have a clear mental model of modern identity systems and authentication flows. You'll understand not just the “how” but also the “why” — why tokens look the way they do, why refresh tokens exist, and why PKCE matters. With these tools, you'll be ready to design authentication systems that balance security, usability, and scalability.

11 JSON Object Signing and Encryption (JOSE)

This chapter covers

- What makes up JavaScript Object Signing and Encryption (JOSE) standard
- Creating and verifying JSON Web Signature (JWS) objects
- Encrypting and decrypting JSON Web Encryption (JWE) objects
- Avoiding common JWS and JWE security pitfalls

We live in a world where data is exchanged between systems implemented in multiple programming languages by multiple teams working for multiple organizations. Systems interoperate using standard networking protocols such as HTTP in a well-defined manner using standard data formats. For example, REST with JSON, SOAP with XML, and gRPC with protocol buffers (protobufs). Standardized data formats for exchanging encrypted and signed data make interoperability significantly easier.

Security protocols such as X.509 digital certificates (chapters 8 and 9), OpenID Connect and OAuth2 (chapters 12 and 13), SAML, TLS (chapter 10) need to exchange encrypted and signed messages. Security protocols rely on standard formats to represent encrypted and signed content. For example, OpenID Connect uses JSON, SAML uses XML, while X.509 certificates are represented using a standardized binary data format.

If you understand the various security data formats, you can easily create and edit files in these formats. More importantly, you will be able to understand error messages and stack traces produced by security libraries that you are coding against. Troubleshooting is simple when you understand the underlying data formats and challenging when you don't. This chapter is a friendly introduction to the widely used JavaScript Object Signing and Encryption (JOSE) suite of standards, along with a basic introduction to JSON Web Token (JWT) standard.

11.1 The Standards Layer Cake

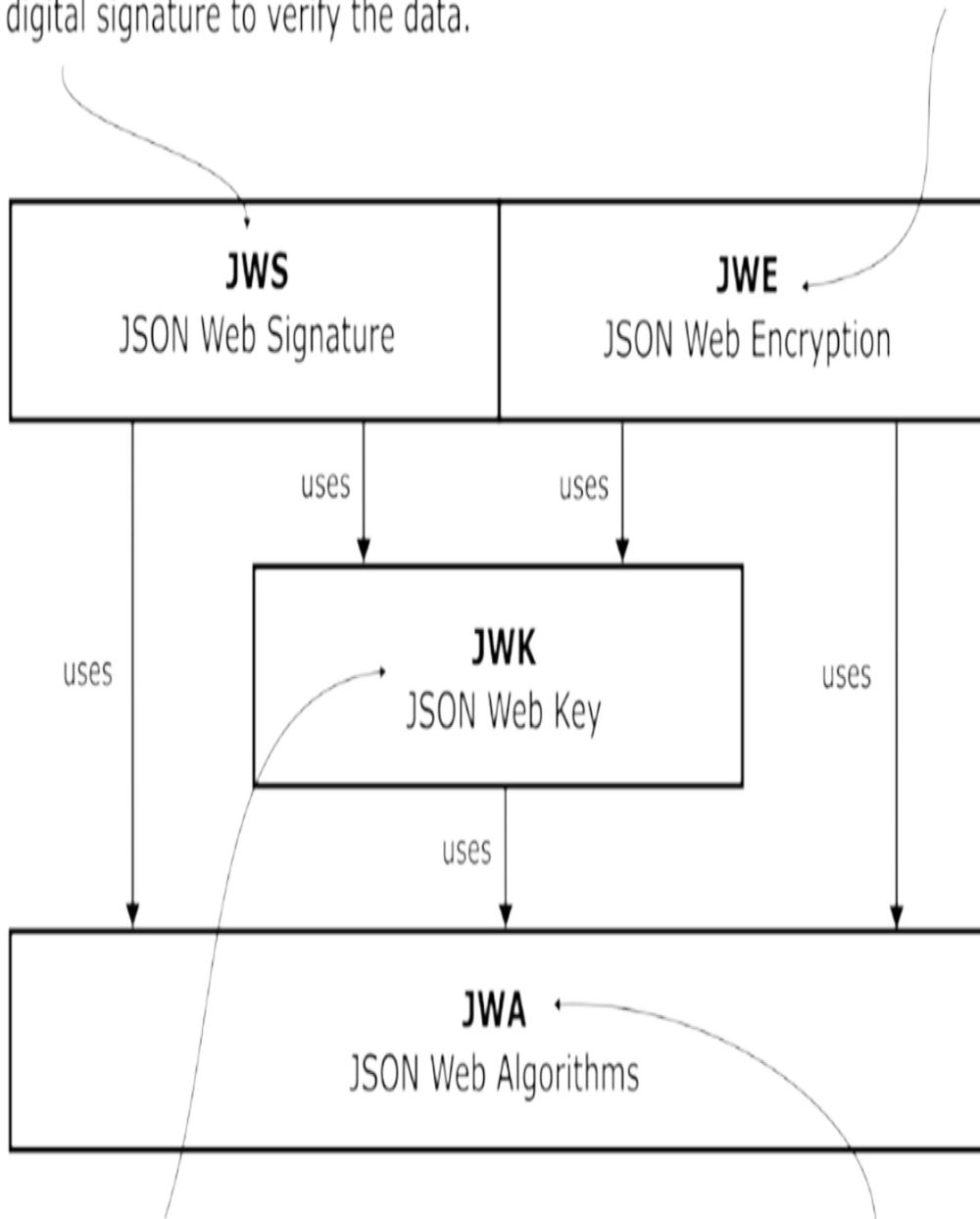
The JavaScript Object Signing and Encryption (JOSE) is a collection of four standards that build on each other (figure 11.1). Each standard focuses on a well-defined set of security capabilities. In this chapter we will explore the four standards that are part of the JOSE suite:

1. *JSON Web Algorithms* (<https://tools.ietf.org/html/rfc7518>) (JWA)
defines string identifiers for commonly used cryptographic algorithms. For example, HMAC based on the SHA-256 hash function which we explored in chapter 4 is identified using the string HS256.
2. *JSON Web Key* (<https://tools.ietf.org/html/rfc7517>) (JWK) represents secret, public, and private keys for various cryptographic algorithms as JSON objects. The JWK standard uses the identifiers defined by JSON Web Algorithms (JWA).
3. *JSON Web Signature* (<https://tools.ietf.org/html/rfc7515>) (JWS)
represents content that must be protected from tampering but does not need to be kept confidential. The JWS standard uses the JSON Web Algorithms (JWA) and JSON Web Key (JWK) standard.
4. *JSON Web Encryption* (<https://tools.ietf.org/html/rfc7516>) (JWE)
represents content that must be protected from tampering and must be kept confidential. The JWE standard uses the JSON Web Algorithms (JWA) and JSON Web Key (JWK) standard.

Figure 11.1 Relationship between the standards that are part of the JSON Object Signing and Encryption (JOSE) suite. JOSE enables interoperability between applications that want to exchange encrypted or signed data using the JSON data format. JOSE standards are used extensively by popular security protocols such as OpenID Connect for Single Sign On.

A standardized way to store
in JSON format some data
and a digital signature to verify the data.

A standardized way to store
encrypted JSON formatted data.



A standardized way to represent
cryptographic keys in a JSON format.

Enables an application to identify
which algorithm it must use to
decode/decrypt/verify data which
was created by a different application.

11.2 The problem solved by JSON Web Algorithms (JWA)

Suppose you receive an encrypted email message and want to decrypt it so you can read it. How do you know which encryption algorithm was used to encrypt the email message? You can assume that the email message was encrypted using AES-GCM-128 because that is the encryption algorithm that you agreed to use with the person who sent you the email. The sending and receiving application can hardcode the choice of algorithm into their implementation. This approach works when you have two applications written by the same team.

If different teams working for different companies write the applications, it is better to add metadata to the encrypted email message indicating what type of encryption algorithm was used to encrypt the email. This enables the receiving application to determine the correct decryption algorithm to use when it receives the message. For example, you can use the string “AES-128-GCM” to identify the encryption algorithm as Advanced Encryption Standard with 128 bits key in Galois Counter Mode (chapter 5).

But another team might choose to use the string “AES-GCM-128” (notice the difference!) to indicate that the encryption algorithm is Advanced Encryption Standard with 128-bit keys in Galois Counter Mode. Using standardized names for encryption algorithms names enables interoperability.

The JSON Web Algorithms is an IETF standard registry of cryptographic algorithm names and identifiers for:

- Digital Signatures - Algorithms used to verify that a message truly comes from the claimed sender and has not been altered.
- Message Authentication Codes - Algorithms that ensure message integrity and authenticity using a shared secret key.
- Key Management - Algorithms that establish, wrap, or exchange cryptographic keys securely between parties.
- Encryption - Algorithms that protect the confidentiality of data by transforming it into unreadable form for unauthorized users.

Some example of standard identifiers are:

- “A256GCM” indicates the advanced encryption standard using a 256-bit key in Galois counter mode.
- “A192CBC-HS512” indicates the advanced encryption standard using a 192-bit key in cipher block chaining mode with an HMAC based on the SHA-2 with a 512-bit output hash algorithm.
- “ES384” indicates the elliptic curve digital signature algorithm using the NISTP-384 curve and SHA-384 HMAC.

To accommodate the addition of new algorithm names, the JWA identifiers are registered with the *Internet Assigned Naming Authority* (IANA). IANA is a non-profit that oversees many critical aspects of the internet, such as the IP address space, root DNS domains, and a list of media content contents. You can see the full list of JWA names at

<https://www.iana.org/assignments/jose/jose.xhtml>. As a developer, you will be able to use the JWA when you are debugging to identify what type of algorithms are being used.

11.2.1 Exercises

1. Why do we need standardized names for cryptographic algorithms?
2. Give two examples of JWA identifiers and explain what they mean.
3. What organization maintains the registry of JWA identifiers?

11.3 JSON Web Key (JWK)

Cryptographic algorithms take a variety of keys as inputs. For example, AES requires a 128, 192, or 256-bit key. Public key encryption algorithms use private and public key pairs, where the public keys are to be shared with many different applications and systems. What is the data format for storing and exchanging key?

In chapter 4, we stored HMAC keys as hex-encoded strings in the Spring Boot application.yml file. This approach works when you have two applications written by the same team, but it doesn't scale for multiple teams spread across multiple organizations. The *JSON Web Key* (JWK) is a

standard for representing keys as JSON objects.

JWK helps represent complex keys that have multiple components; for example, the JSON below represents an elliptic curve public-private key pair using the NIST P-256 curve. We will discuss elliptic curve cryptography in the next chapter.

Listing 11.1 An example of a JSON Web Key

```
{  
  "kty": "EC",  
  "crv": "P-256",  
  "x": "MKBCTNICKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",  
  "y": "4Et16SRW2YiLUrN5vfVHuhp7x8Px1tmWw1bbM4IFyM",  
  "d": "870MB6gfutJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",  
  "use": "enc",  
  "kid": "1"  #A  
}
```

11.3.1 Exercises

4. What problem does JWK solve compared to storing keys as hex strings?
5. What does the "kid" field represent in a JWK?
6. Why is representing keys in JSON format useful in distributed systems?

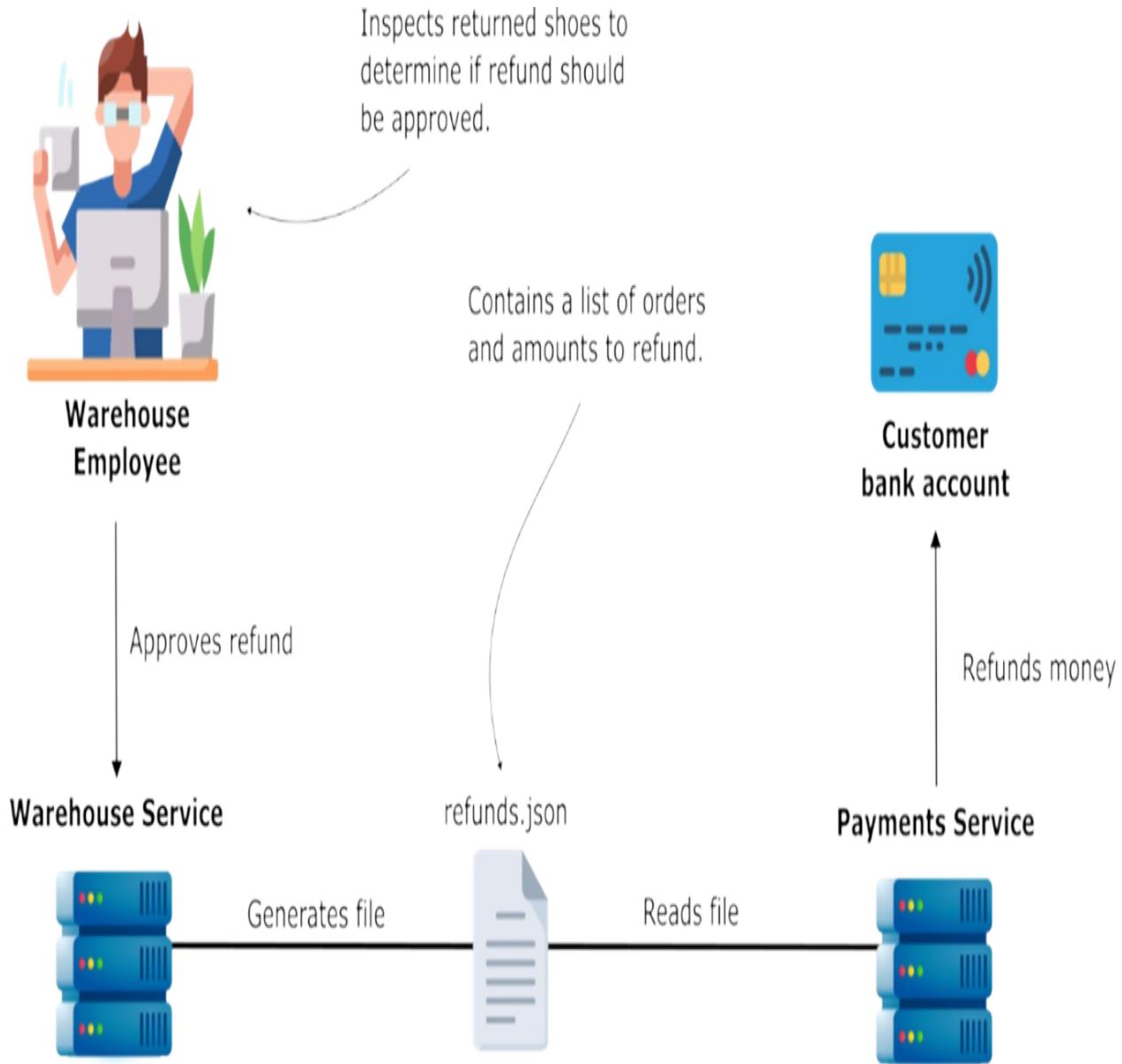
11.4 JSON Web Signature (JWS)

Recall the ACME Inc. online shoe retailer refund processing scenario we used in the previous chapters. We are going to reuse the scenario in this chapter to explore the JSON Web Signature (JWS) standard.

Customers mail shoes they do not like to ACME Inc's warehouse. Warehouse employees verify that the returned shoes are in good condition before authorizing a credit card refund using the warehouse management app. Once a day, the warehouse management app produces a `refunds.json` file. The payments service uses this file to refund customer credit cards (figure 11.2).

Figure 11.2 ACME Inc. staff approve refunds using the warehouse management application.

Once a day the warehouse management application generates a refunds.json file. The payment service refunds customer credit cards for the amount specified in the refunds.json file.



In chapter 4, we used HMAC with SHA-256 to ensure that the refunds.json file was both authentic (created by the warehouse service) and untampered. However, the implementation creates two separate files: refunds.json.hs256 for the HMAC signature and refunds.json for the actual data.

Interoperability between services is a critical architecture design goal for the ACME Inc. software development team (like it should be for any other

software system). Therefore, they want to leverage an industry standard as the data exchange format between the various ACME Inc. services. Is there an industry standard format that can be leveraged to reduce coding effort and improve portability?

JSON Web Signature (JWS) defined by RFC 7515 is an IETF standard “represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures.” (<https://tools.ietf.org/html/rfc7515>) JWS is part of a collection of IETF standards called JavaScript Object Signing and Encryption (JOSE).

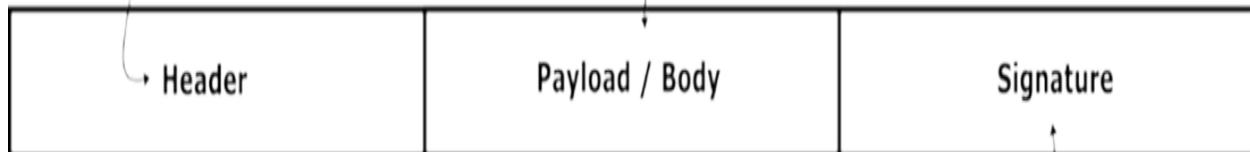
11.4.1 JSON Web Object (JWS) Structure

A JWS object has three parts: header, body, and signature (figure 11.3). The header is a JSON object that describes which cryptographic algorithm used to sign the JWS object. The payload is an arbitrary sequence of bytes. The signature is a message authentication code, or a digital signature based on public key cryptography.

Figure 11.3 The logical structure of a JSON Web Signature (JWS). A JSON metadata header describing the type of signature used. A payload that can be any type of data format not just JSON. Signature to use verify that the header and payload were not tampered with. A JWS payload is always readable to anyone who can access the JWS object.

Base64 JSON formatted data representing metadata (for example, the algorithm used to generate the signature).

The data - might be JSON formatted but can be also a sequence of bytes. Same as the header, the payload (body) is Base64 encoded to ease the storage and transfer.



The signature that ensures that the header and the payload was not tampered with. Its format is Base64 encoded binary.

Take as an example the JSON payload in the next snippet.

```
[ {  
    "orderId" : "12345",  
    "amount" : 500  
, {  
    "orderId" : "56789",  
    "amount" : 250  
} ]
```

This payload turns into the JWS in the next snippet when it is signed using the HMAC with SHA-256. Observe the three parts separated by dots.

eyJhbGciOiJIUzI1NiJ9.WyB7CiAgIm9yZGVySWQiIDogIjEyMzQ1IiwKICAiYW1v

JWS objects are typically exchanged between applications using HTTP header values or URL query parameters. However, the HTTP and URL

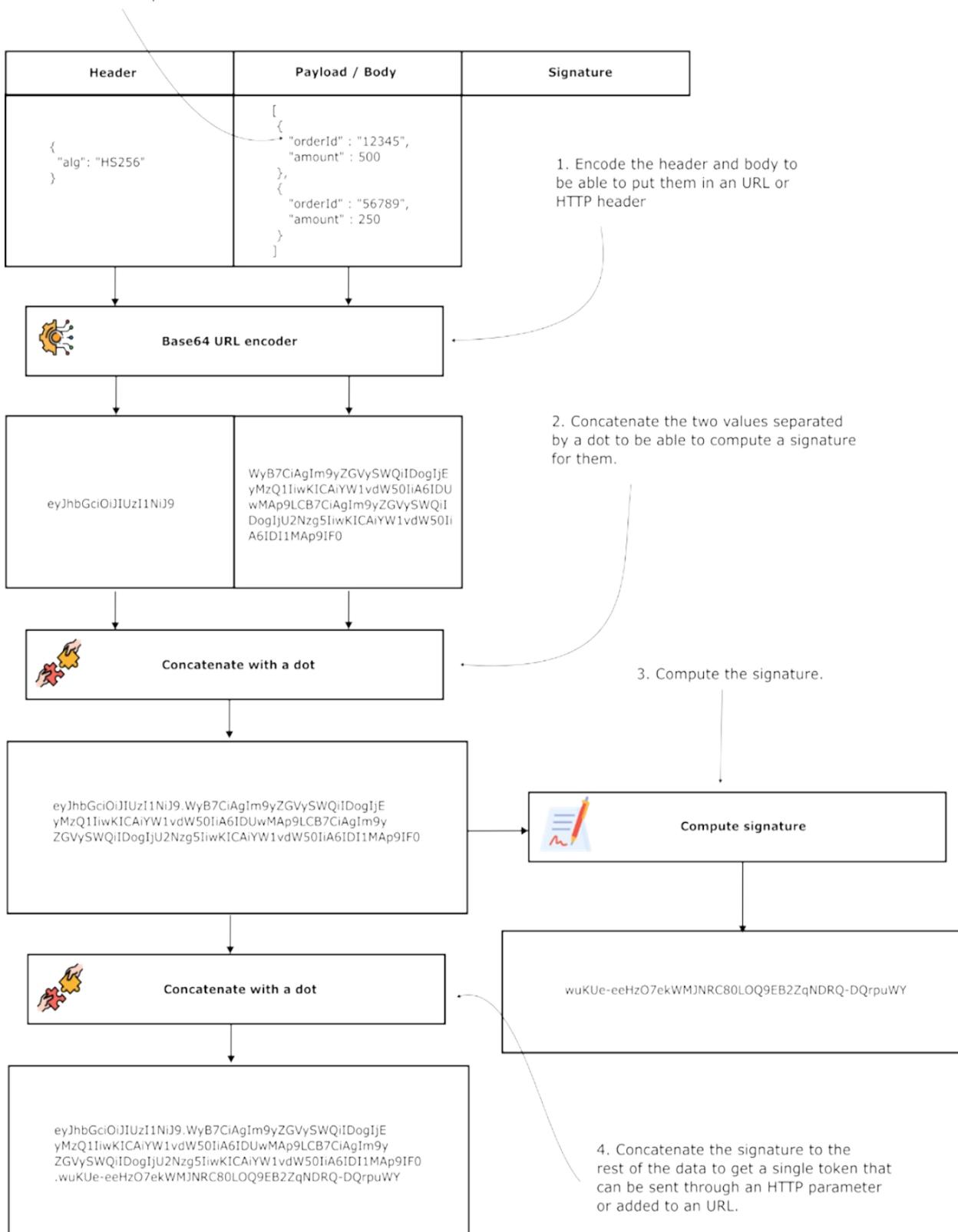
specifications allow a limited set of ASCII characters to be used as header or parameter values. Since a JWS payload can contain arbitrary binary data, it must be encoded in a way that is safe to use in HTTP headers and URL parameters.

To make JWS objects easy and safe to use as an HTTP header and query parameter values, each part of the JWS object is encoded using Base64 URL encoding and then concatenated together and separated by dots.

Diagram 11.4 shows the process of converting the example JSON payload into a JWS.

Figure 11.4 Road from data to a JWS. A JSON Web Signature (JWS) object can be safely embedded in a URL or HTTP headers because it is represented a base64 URL string where each component is separated by a dot.

It is not safe to put these values directly into an URL or into HTTP header values since they contain characters which are not allowed by the URL and HTTP header specifications.



The header of the JWS in figure 11.4 object is eyJhbGciOiJIUzI1NiJ9 decodes to the JSON object in the following snippet.

```
{  
  "alg": "HS256"  
}
```

The “alg” field in the header indicates the type of signature algorithm being used by this JWS object. The algorithm names are standardized by *JSON Web Algorithms* (JWA) in RFC 7518[1]. HS256 is the JSON web algorithm name for HMAC using SHA-256. The payload component of the JWS is shown in the next snippet.

WyB7CiAgIm9yZGVySWQiIDogIjEyMzQ1IiwKICAiYW1vdW50IiA6IDUwMAp9LCB7C

This string decodes to list of orders to refund produced by the warehouse application as shown in the next snippet.

```
[ {  
  "orderId" : "12345",  
  "amount" : 500  
, {  
  "orderId" : "56789",  
  "amount" : 250  
} ]
```

The signature component wuKUe-eeHz07ekWMJNRC80L0Q9EB2ZqNDRQ-DQrpwY is the result of computing the HMAC-SHA-256 on the header and payload. This signature can be decoded to the hex string below

c2e2947be79e1f33bb7a458c24d442f342ce43d101d99a8d0d143e0d0ae9b966

TIP

there are several online tools for working with JWS objects. <https://jwt.io> has a nice *JSON Web Token* (JWT) decoder that can be used to decode the components of JWS object. CyberChef is a handy open-source browser-based

tool produced by the British *Government Communications Headquarters* (GCHQ) for encryption, encoding, compression and data analysis. CyberChef is available at GitHub <https://gchq.github.io/CyberChef/>

You can represent a JWS object as a JSON object with fields rather than the Base64 compact URL discussed above. However, we don't cover the JWS JSON serialization format in this book since you'll not find it used with the various security protocols we discuss in the rest of the book.

Depending on the signing algorithm you use, a JWS can deliver on all the goals of cryptography. The signing algorithm we use in this section does not support the non-repudiation cryptography goal (table 11.1). Remember we discussed a lot over these four objectives in part 2 of this book.

Table 11.1 Cryptography goals and algorithms matrix. JWS can provide non-repudiation if used with public key crypto, which we will show in the next chapter.

Goal	JWS-HS256
Integrity	Yes
Authentication	Yes
Confidentiality	No
Non-repudiation	No

11.4.2 Creating and verifying a JWS object

JSON Web Signature (JWS) is a widely supported standard with implementations available in many programming languages, including Java.

One popular and user-friendly library for working with JWS and other JSON Object Signing and Encryption (JOSE) standards is Nimbus. Nimbus is open-source and commonly used in Java projects, including Spring Security, where it powers support for OpenID Connect and OAuth2 logins. Project ssfd_ch11_ex1 provided with the book demonstrates how a JWS object can be generated in a Java app. We start by adding in the pom.xml file the dependency for Nimbus as shown in the next code snippet.

```
<dependency>
  <groupId>com.nimbusds</groupId>
```

```

<artifactId>nimbus-jose-jwt</artifactId>
<version>9.37.3</version>
</dependency>

```

Listing 11.2 shows the code that creates the JWS object explained in the previous section. Mind that we use a hardcoded secret here to make the example simpler and allow you focus on the important capabilities. But remember that in a real-world app secrets should never be hardcoded or stored in properties files.

Listing 11.2 Create a JWS object and sign it using a secret key

```

public final class JwsUtil {

    private static final byte[] HMAC_SECRET =
        Base64.getDecoder().decode(
            "VGhpc0lzQS1EZW1vLVNlY3JldC1LZXk
             ➔tRm9yLUhTMjU2LUF0TGVhc3QtMzJCeXRlcye="
        );

    public static String generateJws(
        String payloadJson) throws JOSEException {

        JWSHeader header = #A
            new JWSHeader.Builder(JWSAlgorithm.HS256) #A
                .contentType("application/json") #A
            .build();

        JWSObject jwsObject = #B
            new JWSObject(header, new Payload(payloadJson)); #B

        jwsObject.sign(new MACSigner(HMAC_SECRET)); #C

        return jwsObject.serialize(); #D
    }

}

```

To create a JWS signed with HS256, you need two key components:

- Payload: The content you want to include in the JWS. This can be any string, such as JSON data.

- Secret Key: A shared secret used for signing the JWS to ensure its authenticity.

The Nimbus library simplifies working with JWS by providing several useful classes:

- JWSHeader: Represents the header of the JWS. In this case, the header is configured to use the HS256 algorithm for signing.
- Payload: Represents the content of the JWS. This can be any string, such as "Hello, World!" or a JSON object.
- JWSObject: Combines the header, payload, and signature into a single object. The `sign()` method is used to generate the signature. In the example, the method is passed a MACSigner, which is configured with the secret key to compute the HMAC-SHA256 signature.

When an application receives a JWS token, verifying the signature is essential to ensure the token hasn't been tampered with. Verifying the JWS token using the Nimbus library is straightforward, as shown in listing 11.3.

Listing 11.3 Verify a JWS object using Nimbus

```
public final class JwsUtil {

    private static final byte[] HMAC_SECRET =
        Base64.getDecoder().decode(
            "VGhpc0lzQS1EZW1vLVN1Y3J1dC1LZXktR
             ➔m9yLUhTMjU2LUF0TGVhc3QtMzJCeXRlcye="
        );

    public static boolean verifyJws(String compactJws) {
        try {
            JWSObject parsed = JWSObject.parse(compactJws); #A
            return parsed.verify(new MACVerifier(HMAC_SECRET)); #B
        } catch (ParseException | JOSEException e) {
            return false;
        }
    }
}
```

The `JWSObject.parse()` method takes a Base64 encoded string and creates a Nimbus `JWSObject` out of it. The `MACVerifier` class takes a secret key as

input and is used by the `verify()` method to check that the computed HMAC of the JWS matches the signature component of the JWS.

WARNING

The JWS specification includes a potential vulnerability: the “alg” field in the header can be set to none, effectively bypassing signature validation. This means a hacker could intercept a JWS, change the “alg” field to none, modify the payload, and then forward the tampered request. Since `alg:none` is technically a valid option under the specification, some JWS libraries might mistakenly accept the unsigned JWS as valid, allowing the attack to succeed.

Developers must ensure that JWS libraries are configured to reject `alg:none` as a valid option. Unfortunately, many fail to do so, which has resulted in numerous real-world vulnerabilities over the years, despite this being a well-known issue.

For instance, in October 2020, it was revealed that the official UK NHS COVID-19 contact tracing app for Android was affected by the `alg:none` vulnerability. Hackers could potentially exploit this weakness to bypass the app’s security, as detailed in this blog post:

<https://www.zofrex.com/blog/2020/10/20/alg-none-jwt-nhs-contact-tracing-app/>

Adding `alg:none` to the JWS standard was a significant design flaw with serious consequences. This mistake has led several high-profile cryptographers to criticize the JWS and JOSE standards altogether.

Thankfully, the Nimbus MACVerifier, used in the sample code from listing 11.3, throws an exception if `alg:none` is encountered, ensuring the implementation is safe from this vulnerability. Always double-check your JWS/JWT library settings to ensure you use them securely and avoid this known pitfall.

11.4.3 The credit card refunds scenario with JWS

To meet integrity and authenticity, we can use JWS (JSON Web Signature).

- Integrity – the data has not been changed by someone else.
- Authenticity – we can trust who created the data.

The good part is that JWS is an industry standard and has a simple format that can be easily used in Java.

In our example, the Warehouse app now creates a single file called refunds.jws in the root folder of the project using the code we discussed in listings 11.2 and 11.3. You can check this yourself by running the project ssfd_ch11_ex2-warehouse.

Compared to what we did earlier with HMAC in chapter 4, the Warehouse code is much simpler. With HMAC we had to write two files:

- refunds.json (the actual data)
- refunds.json.sha256 (the checksum used to verify it)

With JWS, all this is packed together in one file. The JWS header also contains useful metadata, such as which signing algorithm is used. This makes it easy to change the algorithm or its settings later (for example, switching from SHA-256 to SHA-512).

On the other side, the Payments app (ssfd_ch11_ex2-payments) can open the refunds.jws, check that it really came from the Warehouse app, confirm it was not changed, and then extract the original data (the payload).

To fulfill the exercise you need to follow the steps:

1. Run the Warehouse app to generate the refunds.jws file
2. Take the refunds.jws file and put it in the resources folder of the Payments app.
3. Run the Payments app and observe that the payload is extracted into a new file.
4. Change the secret and play with the refunds.jws file content and observe that the Payments app checks the data was altered and rejects it.
5. Take the content of the refunds.jws file and use jwt.io to decode the content. Observe that the content (both header and payload) can be read (it's not encrypted).

11.4.4 Exercises

7. What are the three main parts of a JWS object?
8. How is a JWS different from the earlier HMAC-based approach in chapter 4?
9. What vulnerability is associated with the "alg": "none" option in JWS?
10. In the Acme Inc. refund scenario, what does JWS guarantee?

11.5 JSON Web Encryption (JWE)

In the previous section, we used the JSON Web Signature to make sure that we could detect tampering in the refunds.json file exchanged between the warehouse and payment services. In this section, we want to encrypt the contents of the refunds.json file to protect confidentiality.

Interoperability between services is a critical architecture design goal for the ACME Inc. software development team. Therefore, they want to leverage an industry standard as the data exchange format between the various ACME Inc services. JSON Web Encryption (JWE) defined by RFC 7516 is an IETF standard that “represents encrypted content using JSON-based data structures. The JWE cryptographic mechanisms encrypt and provide integrity protection for an arbitrary sequence of octets.”[\[2\]](#) (an octet is one byte).

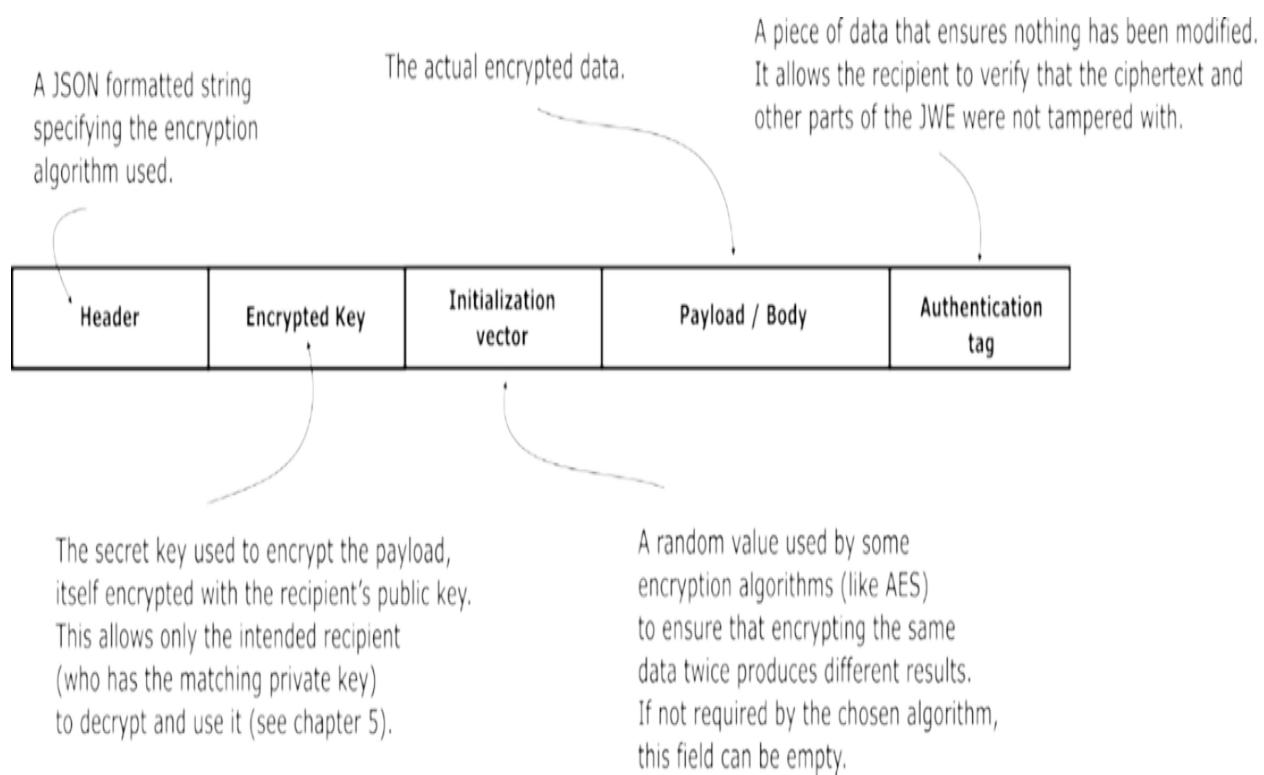
11.5.1 JWE Structure

A JSON Web Encryption (JWE) object is made up of four key components, each serving a specific purpose (figure 11.5):

- Header: Contains metadata about the encryption, such as the algorithm used, the type of encryption key, and any additional parameters needed to decrypt the payload. It essentially provides instructions for how the JWE should be processed.
- Optional Encrypted Key: This is the content encryption key (CEK) that has been encrypted using a public key or shared secret. It allows the intended recipient to securely obtain the CEK, which is used to decrypt the actual data (payload). In some cases, this key may be omitted if a pre-agreed method is used to derive it.

- Payload Cipher Text: The actual content or message being encrypted. This could be anything from a string of text to structured data like JSON. The payload is encrypted using a symmetric encryption algorithm such as AES.
- Authentication Tag: A cryptographic checksum that ensures the integrity and authenticity of the JWE. It verifies that the data has not been tampered with during transmission or storage.

Figure 11.5 Parts of a JSON Web Encryption (JWE) object.



For this example we use the JSON payload in the next snippet.

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "56789",
  "amount" : 250
} ]
```

This input turns into the JWE in the following snippet when it is encrypted

using the AES-GCM using a 256-bit key. If you need a refresher on AES please review chapter 5.

```
eyJ1bmMi0iJBMjU2R0NNIiwiYwxnIjoizGlyIn0.bt9LbkMg4oPbsm-1.CrF8Dq9p
```

Just like a JWS, a JWE can be passed between applications in HTTP headers and URL query parameter so it only uses characters that are legal in URL and http header values. Each part of the JWE object is encoded as a Base64 string separated by a period. The JWE object shown earlier has the initialization vector and authenticated tag bolded so you can easily spot the various components that make up a JWE. Take the value and put it in jwt.io. Unlike the JWS, you will not be able to see the content for this one (since it's encrypted).

This example JWE is encrypted with AES-GCM using a 256-bit key, it does not use the optional encrypted key component which is why there are two periods just before the bolded initialization vector.

The JWE header is described in the next snippet.

```
eyJ1bmMi0iJBMjU2R0NNIiwiYwxnIjoizGlyIn0
```

This header decodes to the JSON object shown in the next snippet.

```
{  
  "enc": "A256GCM",  
  "alg": "dir"  
}
```

The combination of the enc and alg fields provide enough details for an application to decrypt this JWE. There is no encrypted key in this JWE, so there is no content between the second and third dots. The initialization vector is **bt9LbkMg4oPbsm-1**. This initialization vector corresponds to the random sequence bytes given the hex string **6edf4b6e4320e283dbb26fa5**.

The ciphertext is given by the Base64 encoded string presented in the next snippet.

CrF8Dq9pvvzq2grnkI99RtwUpb5geJQ-GdGXzW0_rVunsZr1FDmdtvzYtnV_fcf7R

The plaintext version of the JWE payload ciphertext above is the refunds.json content we have been using since chapter 4 it is shown in the next snippet.

```
[ {  
    "orderId" : "5555555555554444",  
    "amount" : 500  
, {  
    "orderId" : "401288888881881",  
    "amount" : 250  
} ]
```

The authentication tag is given by the string 5vd0ehw4cKTq7iyxXqEvtQ which decodes to a set of bytes given by the hex string “e6f7747a1c3870a4eaee2cb15ea12fb5”.

A JWE object can be represented is a JSON object with fields rather than the Base64 compact discussed above. However, we don’t cover the JWE JSON serialization format in this book since it not used with the various security protocols we discuss in the rest of the book.

11.5.2 Creating and verifying JWE objects

JSON Web Encryption is widely supported with multiple implementations available in Java and other programming languages. Therefore, it is easy to create a JWE in NodeJS application pass it to a Spring Boot Java based microservice that can easily decrypt it using one of the many Java libraries that support JWE.

Nimbus is a popular easy to use open source Java library for working all the various JavaScript Object Signing and Encryption (JOSE) suite of standards. Spring Security uses Nimbus to implement support of OpenID Connect and OAuth2 login. The JWS object explained in the previous section was created the code snippet below. We use project ssfd_ch11_ex3 to demonstrate working with JWEs using Nimbus in a Java app.

Listing 11.4 Create a JWE object encrypted with AES-GCM

```

public String buildCompactJWE(String payload, byte[] keyBytes)
throws Exception {

    JWEHeader header = #A
    new JWEHeader.Builder(JWEAlgorithm.DIR, EncryptionMethod.A256
        .contentType("application/json")
        .build();

    JWEObject jweObject = #B
    new JWEObject(header, new Payload(payload));

    jweObject.encrypt(new DirectEncrypter(keyBytes)); #C

    return jweObject.serialize(); #D
}

```

To create a JWE object with a payload encrypted with AES GCM with a 256-bit key is easy using Nimbus classes for working with JWE.

- `JWEHeader` is the used to Java representation of a JWE header.
- `Payload` is used to represent the clear text payload that will be inserted into the JWE
- `JWEObject` is a combination of a header and payload it contains methods to encrypt, serialize, pares, and decrypt, the palyoad.
- `DirectEncrypter` is Nimbus object that calls the java Cryptography libraries to perform encryption, it hides the complexity of correctly using the `javax.crypto` libraires.

Notice that the code in listing 11.4 is much simpler than using the `java.crypto` library directly as we did the second part of this book. We don't have to worry about generating an initialization vector or configuring the AES-GCM algorithm as those are done by the Nimbus framework when we selected the `EncryptionMethod.A256GCM` configuration for the JWE encryption algorithm.

We also don't have to write any code to output the metadata required to decrypt the ciphertext later. Another benefit is that we don't have to document the data format for any consumer wishing to decrypt the JWE. The code to decrypt the JWE is shown below.

Listing 11.5 Decrypt a JWE object encrypted with AES-GCM

```
public String decryptCompactJWE(String compactJWE, byte[] keyBytes  
throws Exception {  
  
    JWEObject jweObject = JWEObject.parse(compactJWE); #A  
  
    jweObject.decrypt(new DirectDecrypter(keyBytes)); #B  
  
    Payload payload = jweObject.getPayload(); #C  
    return payload == null ? null : payload.toString(); #C  
}
```

The `JWEObject.parse()` method takes standard JWE Base64 encoded string and turns it into Java object. The `DirectDecrypter` class takes care of decrypting the payload ciphertext. Listing 11.6 shows how we call both encrypt and decrypt operations in the Main class to demonstrate how they work.

Listing 11.6 Demonstrating the encryption and decryption for JWE

```
public class Main {  
    public static void main(String[] args) {  
  
        try {  
            ResourceReader reader = new ResourceReader(); #A  
            String json = reader.readResourceAsString("refunds.json");  
  
            byte[] keyBytes = new byte[32]; #B  
            new SecureRandom().nextBytes(keyBytes);  
  
            JweBuilder jweBuilder = new JweBuilder(); #C  
            String compactJWE = jweBuilder.buildCompactJWE(json, keyBytes)  
  
            System.out.println(compactJWE); #D  
  
            String decrypted = #E  
                jweBuilder.decryptCompactJWE(compactJWE, keyBytes);  
  
            System.out.println(decrypted); #F  
        } catch (Exception e) {  
            e.printStackTrace(System.err);  
        }  
    }  
}
```

When we talk about cryptography, we usually think about four main security goals: making sure data is not changed by accident or on purpose (integrity), checking who created or sent the data (authentication), keeping the content secret (confidentiality), and preventing someone from denying they sent the data (non-repudiation). Different algorithms help us reach these goals in different ways. The table below shows how JWS with HMAC (HS256) and JWE with direct encryption (DIR) meet these goals.

Table 11.2 Cryptography goals and algorithms matrix. JWE and JWS can provide non-repudiation if used with public key crypto which we will show in the next chapter.

Goal	JWS-HS256	JWE DIR encryption
Integrity	Yes	Yes
Authentication	Yes	Yes
Confidentiality	No	Yes
Non-repudiation	No	No

11.5.3 Exercises

11. What additional cryptographic goal does JWE provide compared to JWS?
12. List the four components of a JWE object.
13. In the example given, why is the encrypted key part missing?
14. What is the role of the authentication tag in JWE?

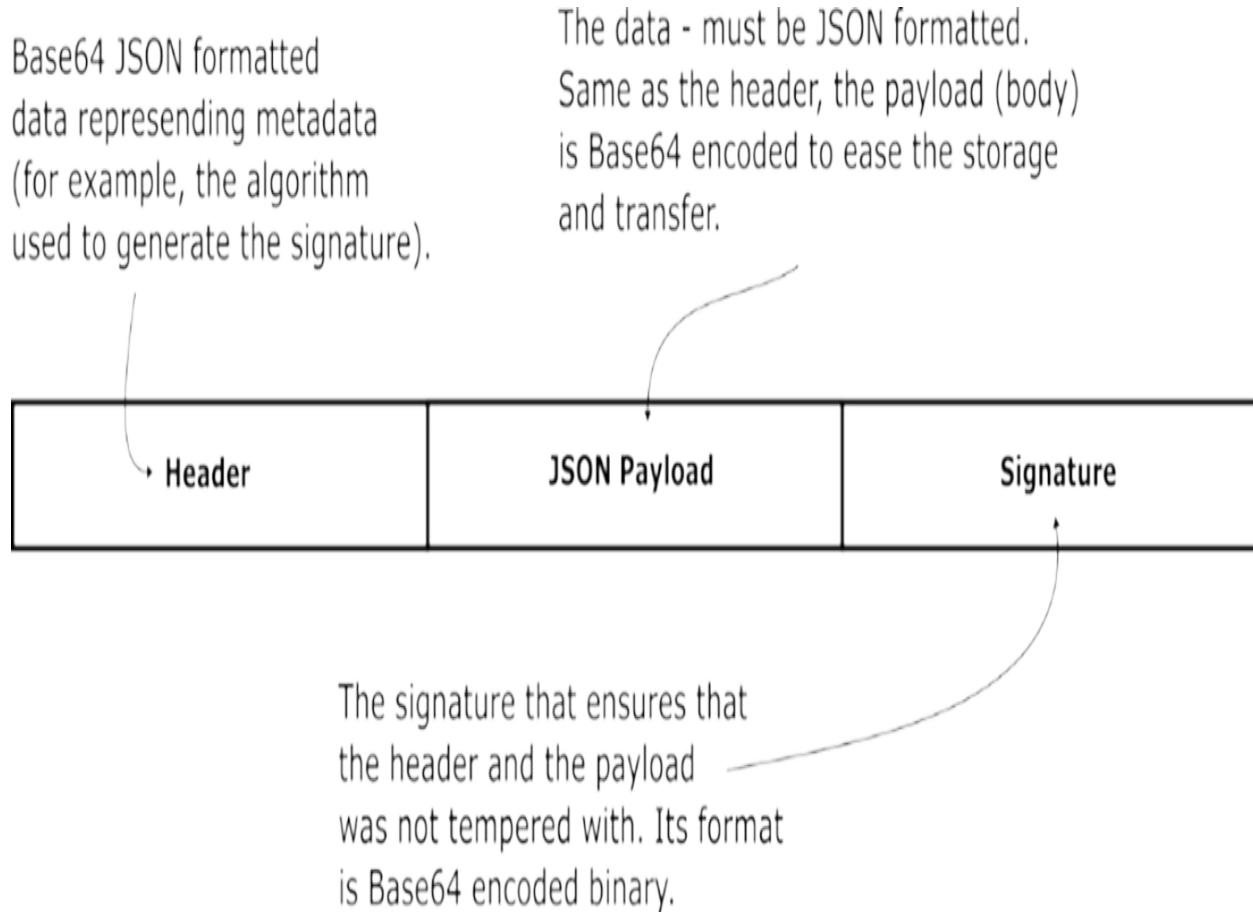
11.6 JSON Web Token (JWT)

The JSON Web Token (<https://tools.ietf.org/html/rfc7519>) (JWT) is standard for signed or encrypted security tokens which is used by OpenID Connect. We'll talk a lot about the OpenID Connect protocol and OAuth2 (the framework on which OpenID Connect relies) in chapter 12 and 13. A JWT can be either a JSON web signature object or a JSON web encryption with the following requirements.

- The payload must be a JSON object
- The payload JSON can have standard fields which are called claims such as sub which is defined to be the id of the user that the token

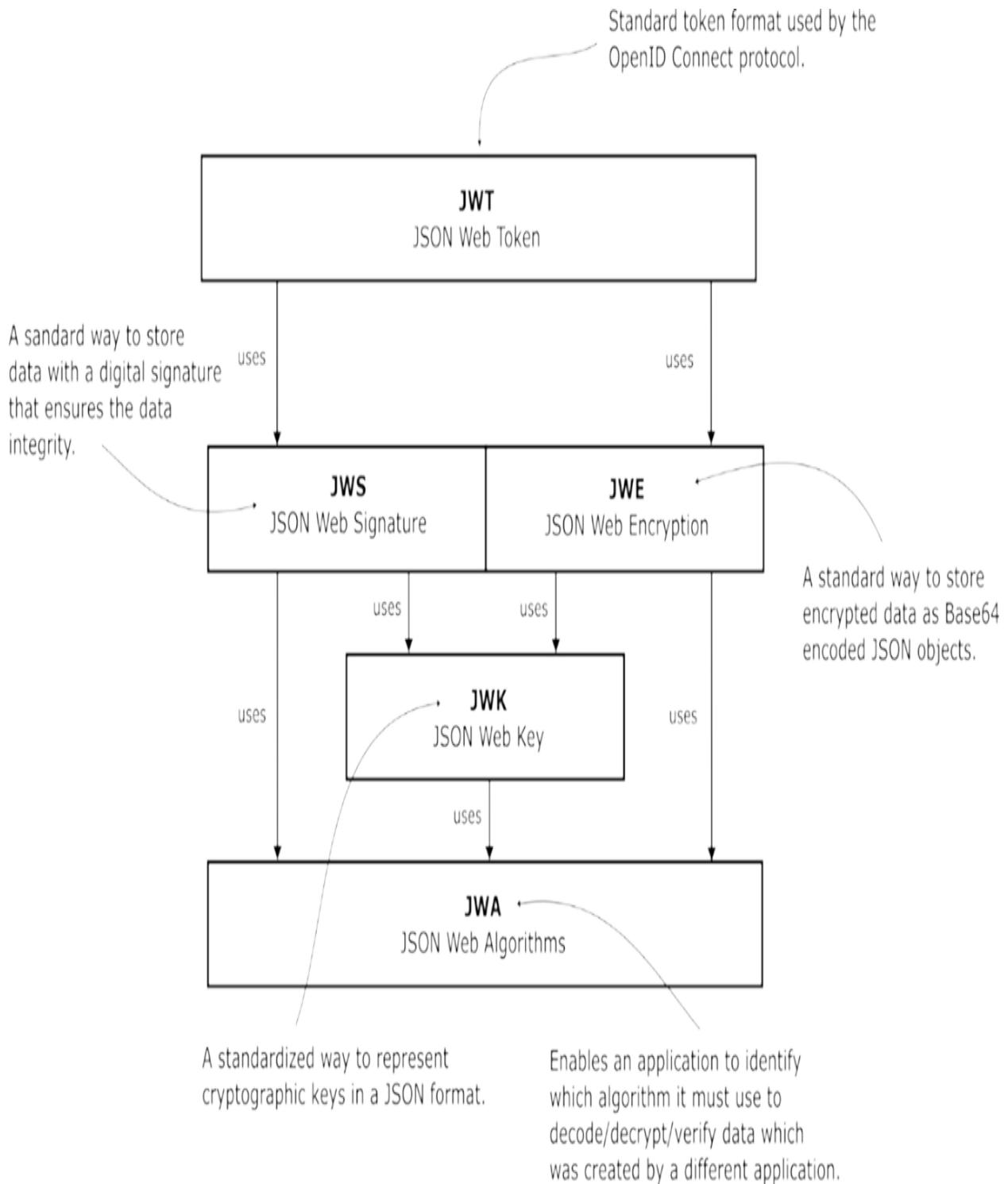
belongs to.

Figure 11.6 A JSON web token (JWT) can be a JWS object with the added restriction that the payload must be JSON object.



The primary difference between a JWT and a JWE or JWS object is that the JWT must have JSON payload. JWE and JWS can have any type of data as a payload. The relationship between JWT, JWE, JWS, JWK, JWS is illustrated in figure 11.7.

Figure 11.7 The collection of standards. JSON Web Token (JWT) builds on top of the JWE and JWS standards. JWT is the token format for the OpenID Connect standard.



Our coverage of JWT, JWS, and JWE used algorithms that require a secret to be shared between the producer and consumer of the JWT. In the next chapter we will learn about public key cryptography and look at how we can use JWS and JWE without the overhead of needing to share a key. Computer

security is a complex challenging topic to learn. We will continue breaking security down into digestible topics.

Common criticisms of JWS, JWE, JWT

The JOSE and JWT standard have come up in for heavy criticism by security experts. The main point of criticism is that JWT, JWS, and JWE standard provide too many configuration options some of which disable security features such as signature validation. This makes JWT, JWS, and JWE easy for developer to use incorrectly. Too much flexibility in a security standard leads to complex implementations that can be buggy with security vulnerabilities

For example, The JWS specification allows the alg field in the header to be set to the value none thus by-passing signature validation completely. A hacker can intercept a JWS change the alg field to none, modify the payload and forward the request. Since alg none is a valid option many JWS libraries accept the unsigned JWS as valid thus bypassing scrutiny.

Developers must configure JWS libraries to reject “none” as a valid alg option. However, many don’t, and this has led to numerous real-world vulnerabilities over the years even though this problem is well known.

For example, in October 2020 it was disclosed[\[3\]](#) that the official UK NHS COVID-19 contact tracing app for Android suffered from alg:none vulnerability. Adding alg “none” to the JWS standard was a serious design mistake with catastrophic consequences. This design mistake is the reason why several high-profile cryptographers dislike the JWS and JOSE standards. The IETF has released new RFCs to provide guidance on how to use JOSE correctly it is important that you use a library that implements best practices. Several alternatives to JOSE have been proposed but none have yet gained widespread use or become official standards. As developer it is important that you understand JOSE and how to use it safely.

11.6.1 Exercises

15. How is a JWT related to JWS and JWE?

16. What restriction does a JWT payload have compared to JWS or JWE?
17. Give an example of a standard claim in a JWT.
18. What is one major criticism of the JWT/JWS/JWE standards?

11.7 Answers to exercises

1. Why do we need standardized names for cryptographic algorithms?
Standardized names ensure interoperability between applications built by different teams and prevent confusion caused by inconsistent naming.
2. Give two examples of JWA identifiers and explain what they mean
Example 1: A256GCM means AES encryption with a 256-bit key in Galois Counter Mode. Example 2: ES384 means an Elliptic Curve Digital Signature Algorithm using the NIST P-384 curve and SHA-384.
3. What organization maintains the registry of JWA identifiers?
The Internet Assigned Numbers Authority (IANA) maintains the JWA registry.
4. What problem does JWK solve compared to storing keys as hex strings?
JWK provides a standard JSON-based way to represent complex keys, making them easier to share between applications and organizations.
5. What does the "kid" field represent in a JWK?
"kid" stands for Key ID and uniquely identifies the key.
6. Why is representing keys in JSON format useful in distributed systems?
JSON is a widely supported and human-readable format, which makes it easier to exchange and use keys across different platforms and languages.
7. What are the three main parts of a JWS object?
The header (algorithm and metadata), the payload (the data), and the signature.
8. How is a JWS different from the earlier HMAC-based approach in chapter 4?
JWS wraps the payload and signature into a single standardized format, making the exchange easier and more interoperable than managing separate files.
9. What vulnerability is associated with the "alg": "none" option in JWS?
If "alg": "none" is accepted, it bypasses signature validation, allowing attackers to tamper with the payload undetected.
10. In the Acme Inc. refund scenario, what does JWS guarantee?

JWS guarantees integrity (the data wasn't changed) and authenticity (it came from the warehouse).

11. What additional cryptographic goal does JWE provide compared to JWS?

JWE provides confidentiality by encrypting the payload.

12. List the four components of a JWE object.

Header, encrypted key (optional), ciphertext (payload), and authentication tag.

13. In the example given, why is the encrypted key part missing?

The example uses alg: "dir" (direct encryption), which means both parties already share the content encryption key, so no encrypted key is needed.

14. What is the role of the authentication tag in JWE?

The authentication tag ensures that the encrypted data has not been tampered with.

15. How is a JWT related to JWS and JWE?

A JWT is either a JWS or JWE with the added requirement that the payload must be a JSON object.

16. What restriction does a JWT payload have compared to JWS or JWE?

JWT payloads must always be JSON, while JWS/JWE payloads can be any type of data.

17. Give an example of a standard claim in a JWT.

The sub claim, which identifies the subject (e.g., user ID) of the token.

18. What is one major criticism of the JWT/JWS/JWE standards?

The standards are considered overly flexible, offering insecure options like alg: none, which can lead to vulnerabilities if misused.

11.8 Summary

- JSON Object Signing and Encryption (JOSE) is suite of standards used to represent cryptographic algorithms, keys, signed content, and encrypted content as JSON objects.
- JOSE is widely used with excellent support in many programming languages. However, it has come in some criticism due to unnecessary complexity in the standard that make it easy to misuse. Watch out for the JWS alg:none vulnerability in any application or library you are using.

- JSON Web Signature (JWS) is an industry standard data format for signed data, it has JSON metadata header, a payload that can have any format, and a signature to validate the payload and header were not tampered with.
- JWS support signatures with message authentication codes (MACs) which we covered in this chapter and digital signatures which we will cover in a future chapter.
- JSON Web Encryption (JWE) is an industry standard data format for representing encrypted data in JSON. It supports AES and has a lot of implementations in different programming languages.
- JSON Web Key (JWK) used to represent cryptographic keys as JSON objects.
- JSON Web Algorithm (JWA) used to define the various algorithms used by JWS, JWE, and JWK.
- Always consult with your Information Security team to make sure you are using corporate recommended configurations of common cryptographic algorithms.
- The examples in this book are optimized for educational value, they take shortcuts to make the code fit on the page, and to emphasize the concepts. Don't copy and paste the sample code blindly, you must make it production ready before you use it.

[1] <https://tools.ietf.org/html/rfc7518>

[2] <https://tools.ietf.org/html/rfc7516>

[3] <https://www.zofrex.com/blog/2020/10/20/alg-none-jwt-nhs-contact-tracing-app/>

12 Single Sign On (SSO) using OAuth2 and OpenID Connect

This chapter covers

- What is Single Sign On (SSO)
- Applying OAuth2 and OpenID Connect

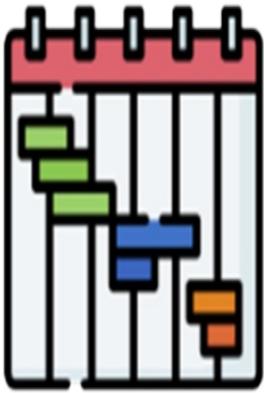
Suppose that over the years, Acme Inc. has experienced phenomenal growth, expanding both its business and customer base. As the company flourished, so did its number of employees and the demand for its services. Both employees and customers rely on various applications provided by the enterprise to interact with the company.

As an employee, managing multiple sets of credentials for different applications throughout the workday is inconvenient and inefficient. Likewise, customers prefer the convenience of using their existing social media login credentials when accessing the Acme Inc. online store.

Simple applications may manage user credentials independently. However, as systems expand and become more complex, strategies for handling authentication and authorization must evolve accordingly. In this chapter, we will explore how large-scale systems implement authentication and authorization effectively.

Figure 12.1 illustrates Jeanny, an employee at ACME Inc.

Figure 12.1 Jeanny uses multiple apps as part of her daily work routine, requiring her to authenticate into each one individually every day.



manages time

operates transactions

manages tasks

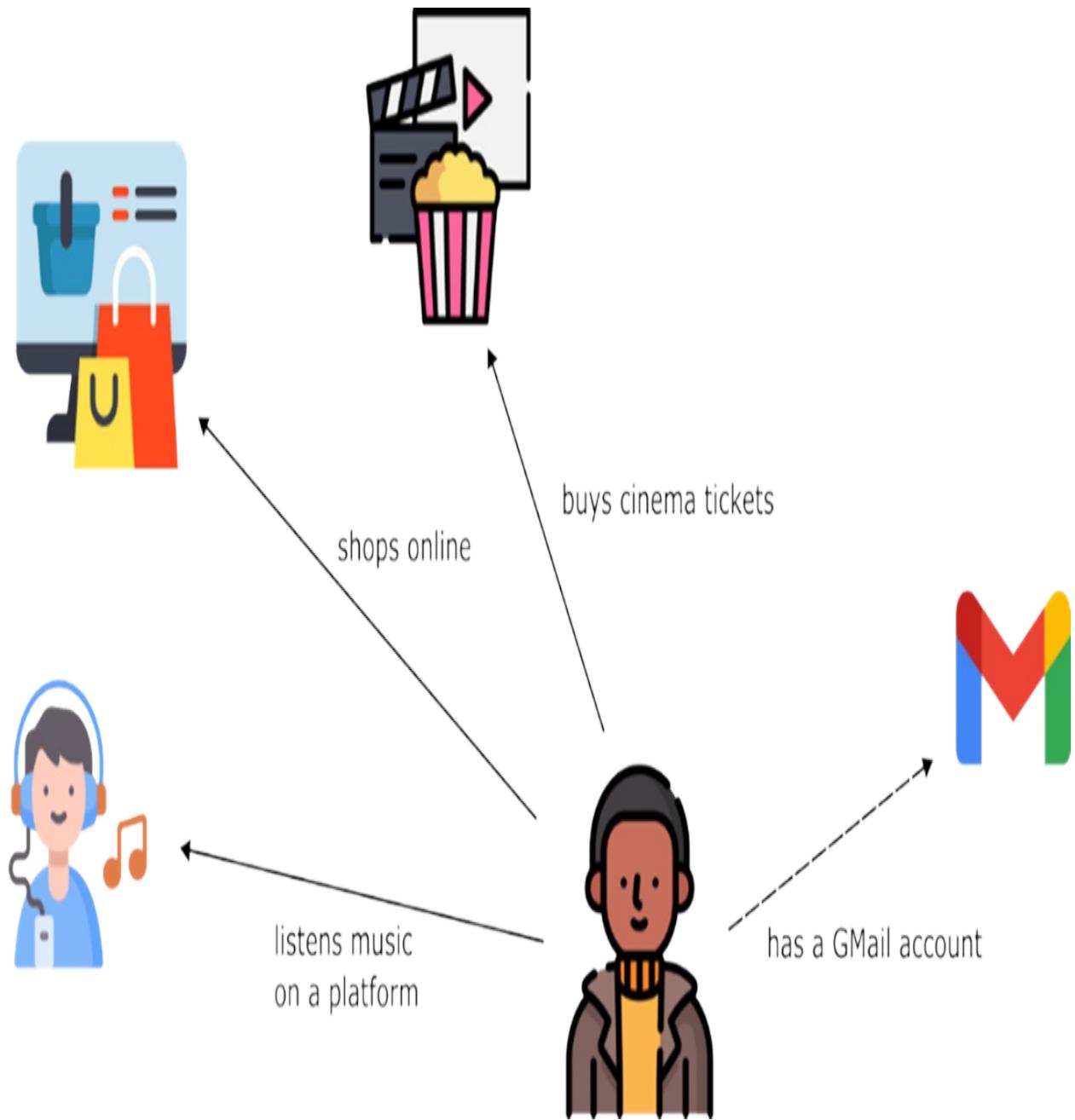
works with documents



Throughout her workday, Jeanny uses several applications to manage documents, transactions, work hours, and more. She finds it inefficient to log in separately to each app and would prefer a single set of credentials, allowing her to avoid remembering multiple usernames and passwords for the various apps she uses.

Figure 12.2 shows Patrick, a typical customer of ACME Inc.

Figure 12.2 Patrick shops online, listens to music on various platforms, and purchases tickets for shows and movies, all using different apps and websites. Since he already has a Gmail account, he prefers to authenticate across these services using his Gmail credentials.



Like many others, he uses multiple platforms to shop online, buy tickets, watch movies, and more. Patrick prefers a centralized way to manage his credentials, eliminating the need to remember different usernames and passwords for each app. Since he already has a Gmail account, he would like to use it to authenticate across all these platforms.

In this chapter, we'll help both Jeanny and Patrick. In section 12.1, we'll discuss how to separate the authentication responsibility in an identity

provider. An identity provider is an application that manages credentials and supports other apps that authenticate individuals.

Once you have a solid understanding of this design, we will dive into two of the most commonly used authentication flows—the authorization code flow and the client credentials flow—in section 12.2. In section 12.3 we'll apply this new approach to our demonstrative application Acme Inc.

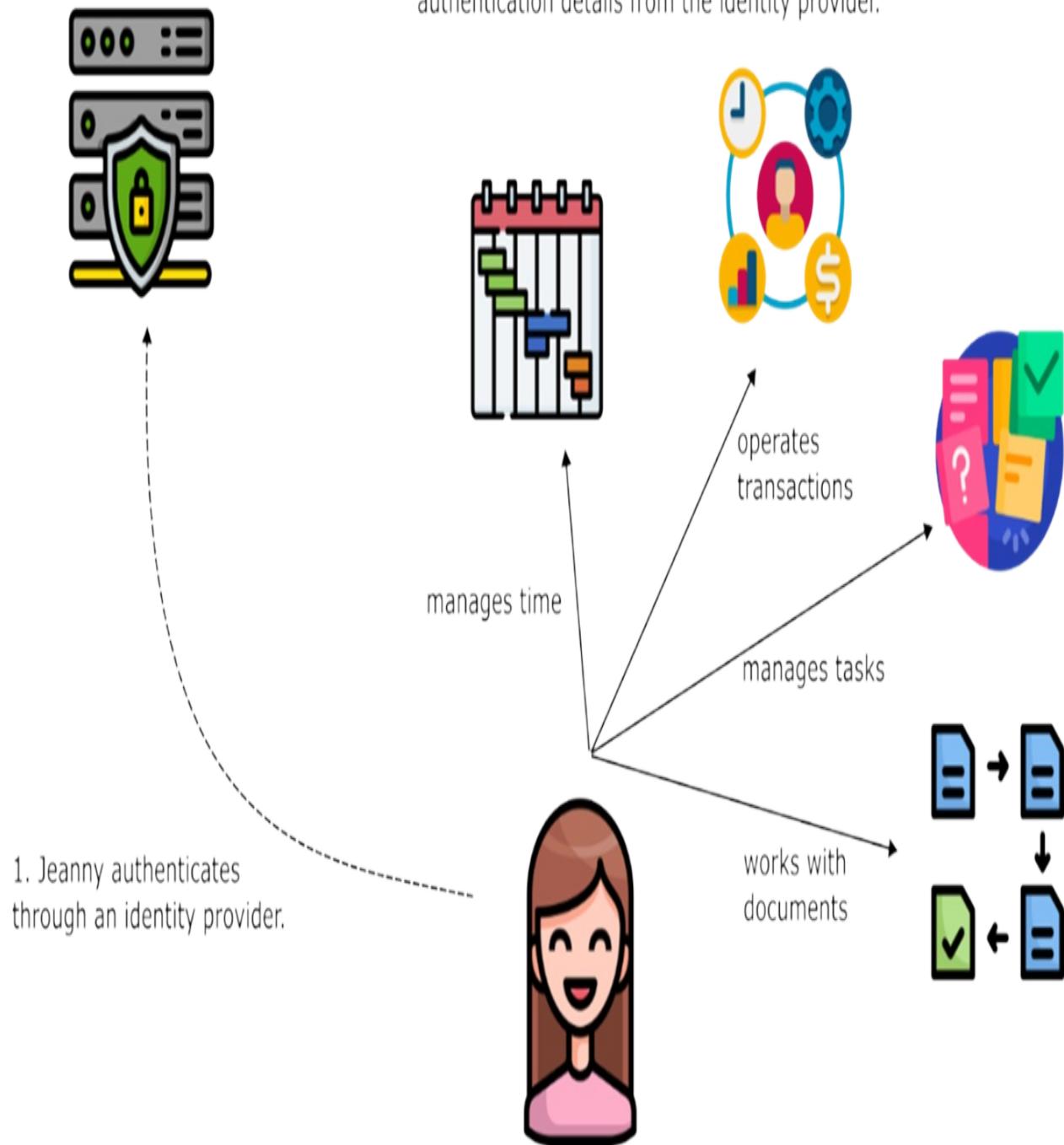
12.1 Splitting security data management responsibilities

In this section, we'll explore how to decouple authentication from the other responsibilities of your app. By doing this, multiple apps can rely on a single, trusted app for authentication. This approach aligns with Jeanny and Patrick's goals, enabling users to authenticate through a unified, trusted app.

Figure 12.3 figure illustrates how we separate the authentication responsibility into another app, which we'll now call the Identity Provider (IdP).

Figure 12.3 Before accessing an app, the Identity Provider (IdP) authenticates the user. The IdP handles the authentication process, while the other apps rely on it to provide the necessary details for authorizing the user to use the specific features they offer.

2, Jeanny can now use the apps without needing to authenticate individually for each one. The apps retrieve the necessary authentication details from the identity provider.



1. Jeanny authenticates through an identity provider.

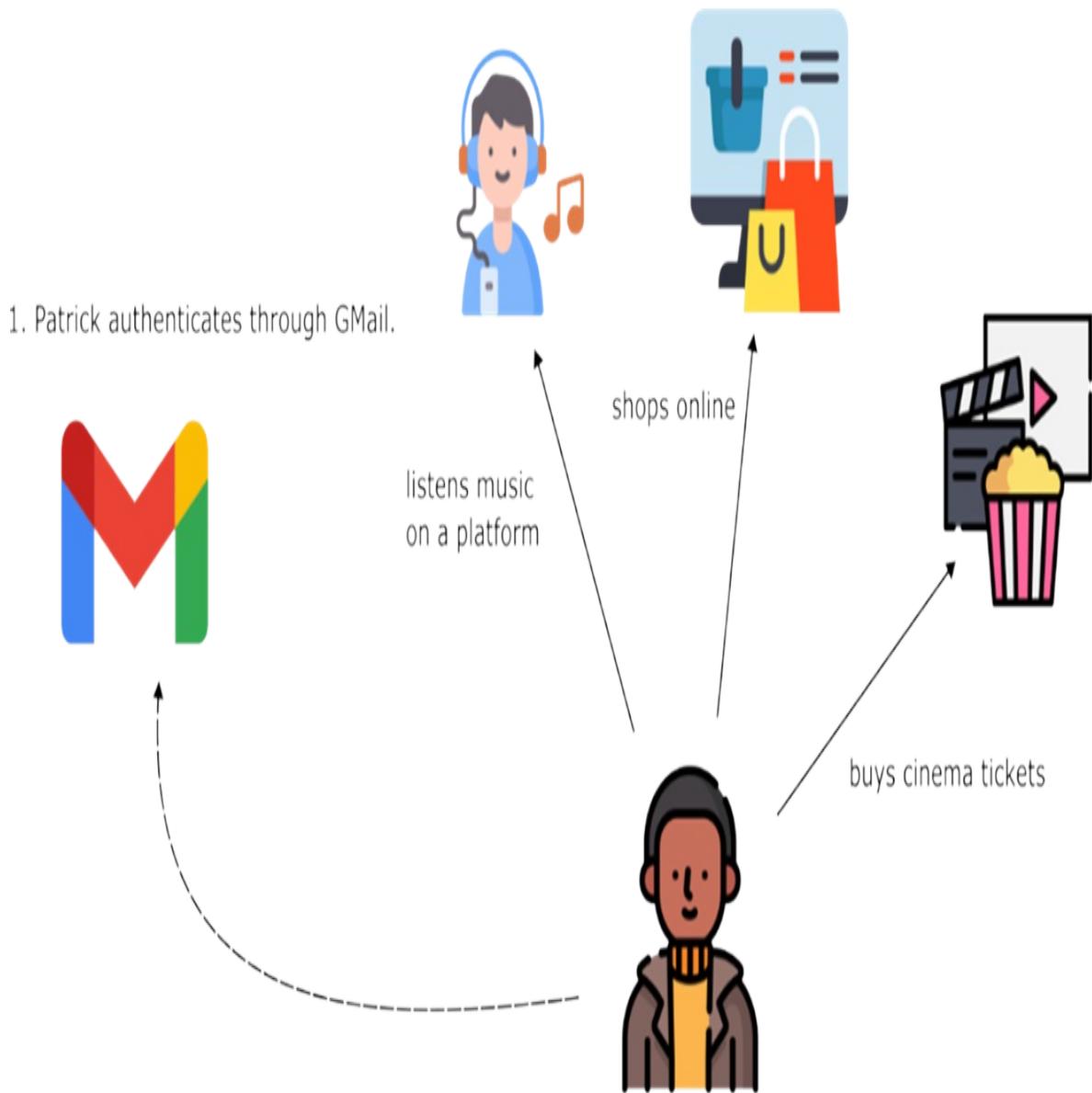
In some cases, people may also refer to it as an Authorization Server (AS). However, to stay consistent throughout this book, we'll continue to call it the Identity Provider.

In the figure, you can see the two steps the user follows to authenticate and access either app:

1. The user authenticates through the IdP, where we've centralized the authentication capabilities.
2. The user then accesses either app, already authenticated by the IdP.

We'll address Patrick's concerns similarly. In his case, Gmail will serve as the IdP. The app Patrick uses must be designed to offer users multiple authentication options. Typically, apps allow authentication via various social media platforms or email services, such as Facebook, X, Google, LinkedIn, and others (figure 12.4).

Figure 12.4 Authenticating using a known social media account. Patrick uses his Google credentials to sign into multiple apps. In this case, Google acts as the IdP. The other apps Patrick uses are designed to let users, like him, authenticate with their Google credentials. Generally, apps offer multiple authentication options, including social media platforms and email services.



2. Patrick can now use his favorite online services.
These services will identify Patrick through GMail
which is now his identity provider.

But how does this flow actually work? How does one app know that another app has properly authenticated a user, and how does it retrieve the necessary details about that user? By separating the authentication responsibility into a different app, we also need a mechanism for the authentication service to communicate with the other apps that rely on it.

Let's start by discussing the most common approach to implementing this

authentication design: using tokens and cryptographic keys. You might want to take a moment to review chapters 6 and 7, where we covered JOSE standards and asymmetric key encryption. What we'll discuss next builds directly on those concepts.

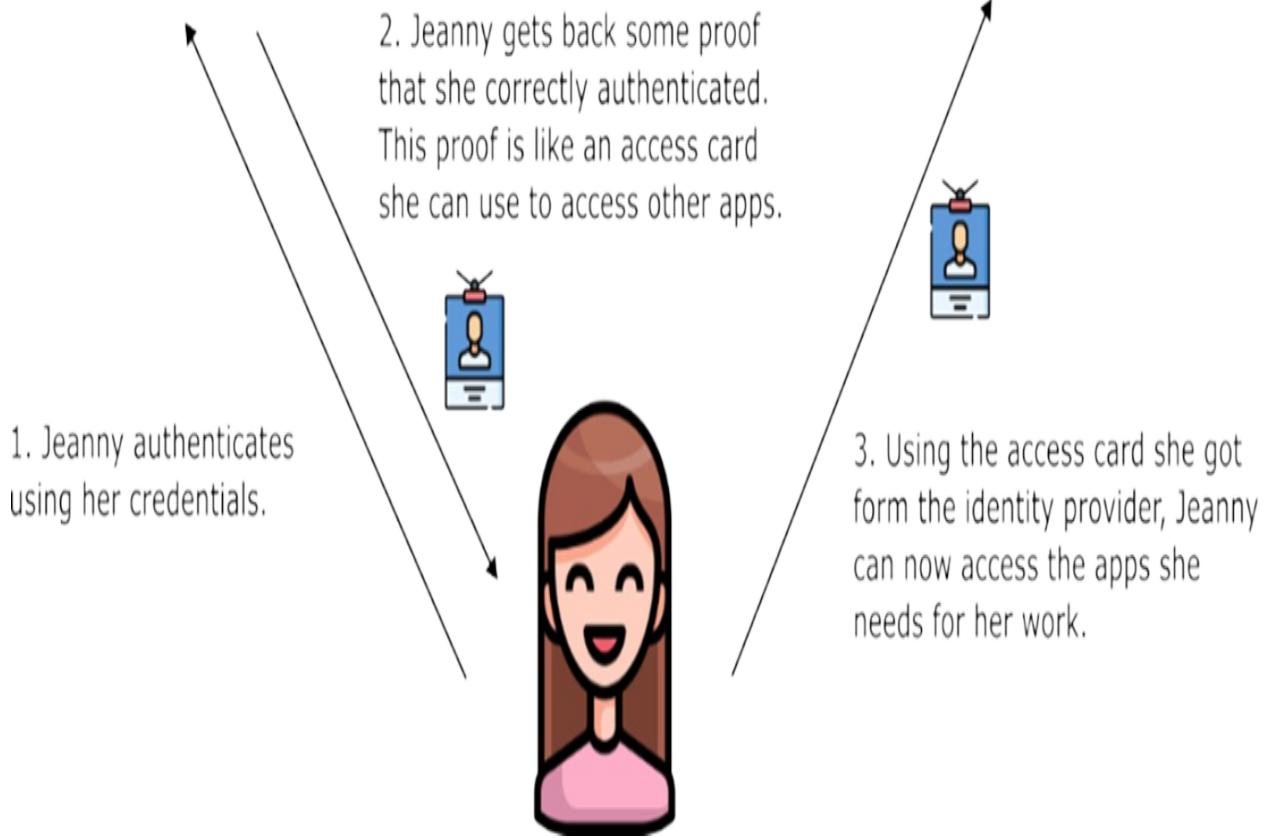
After we cover the most common approach, I'll provide details on other variations of this design that work slightly differently. We'll also compare these methods so you'll know when to apply each in real-world scenarios.

For now, let's focus on tokens and keys. Refer to figure 12.5, which outlines the three steps that occur when Jeanny wants to use a feature in her time management application for her daily tasks. Since she's just starting her workday, she hasn't had a chance to authenticate yet. Her time management app redirects her to the company's IdP, which handles authentication for all the common apps used by employees.

The following happen:

1. Jeanny authenticates with the IdP using her credentials.
2. The IdP provides a token that takes the role of an access card.
3. The app uses the token (access card) to prove to the time management app that she has authenticated, providing all the necessary details the app needs for authorization.

Figure 12.5 The user authenticates with the IdP, which then issues a token. This token serves as proof of the user's identity and successful authentication.



All right! I can already imagine the questions running through your mind:

- How long can the access card (token) be used?
- How does the time management app verify that the token is valid?
- Can someone steal the token and misuse it?

Don't worry, I've got you covered! We'll build on what we've discussed so far, and by the end of this chapter, all these questions will be answered.

12.1.1 Exercises

1. What is Single Sign-On (SSO)?
2. What is an Identity Provider (IdP)?
3. Why do we separate authentication into its own app?

12.2 Using authentication flows

Let's dive deeper into how authentication works to fill in the gaps and give you a complete understanding of this design. In this section, we'll cover OAuth 2, OpenID Connect, and the different authentication flows (also known as grant types). This will provide you with a clear picture of how one app can rely on another to authenticate its users.

To keep things simple, think of OAuth 2 as a set of guidelines (or more formally, a specification) for implementing authentication and authorization. You can find the full specification here:

<https://datatracker.ietf.org/doc/html/rfc6749>. OpenID Connect is a protocol built on top of OAuth 2, and its details are outlined here:

https://openid.net/specs/openid-connect-core-1_0.html. But before you feel overwhelmed, start crying, and spiral into a rage that could destroy the universe, let me walk you through the most essential details in this section.

We've already covered Identity Providers (IdPs) and the separation of authentication from your app's other responsibilities in section 12.1. In addition to the IdP, the OAuth 2 specification also introduces the concept of a resource server. The resource server is simply a fancy term for your app's backend, which handles all the business logic and needs to identify the user in order to apply the appropriate security restrictions through authorization.

Lastly, we introduce the client. The client is the app that users interact with directly. This could be a webpage running in your browser that connects to a backend (resource server) to handle data, or it could be a mobile app running on your iOS or Android device.

These four are the main actors that interact with one another to define the authentication flow:

- The user – person who uses the application

- The client – application used directly by the user such as a page in a web browser or a mobile application
- The IdP – the server responsible for authentication and user and client details management.
- The resource server – the backend application whose resources (usually endpoints) are consumed by the client.

In section 12.2.1, we discuss the most used OAuth 2 authentication flow. We name this flow, the authorization code flow or authorization code grant type.

12.2.1 The authorization code grant type

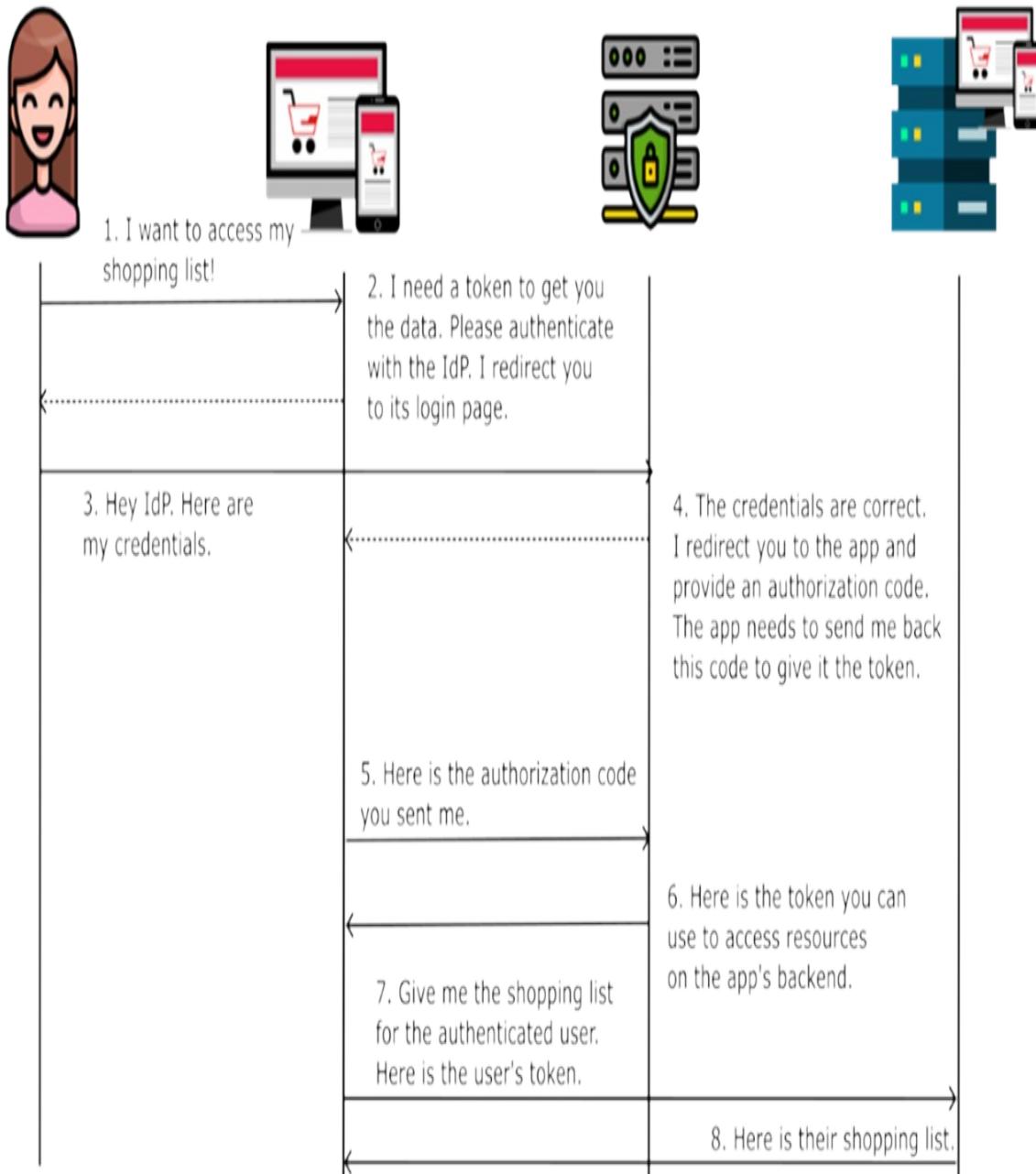
The authorization code authentication flow is the most commonly used OAuth 2 flow. In OAuth 2 terminology, an authentication flow is referred to as a "grant type," so I'll continue using that term in this chapter. A grant type is essentially a sequence of steps that allows a client app to access backend data on behalf of a user or independently (when no user is involved).

In this section, we'll discuss the authorization code grant type, which is used when a user is involved. Then, in Section 12.2.3, we'll cover the client credentials grant type, which allows an app to authenticate without user involvement. In between these two sections, we'll also discuss in more detail what access tokens are and how they are used as access cards in section 12.2.2,

Figure 12.6 illustrates the steps for the authorization code grant type. In the diagram, you'll see the four main OAuth 2 actors: the user, the client, the IdP, and the resource server. Let's imagine a scenario where our user, Jeanny, wants to view her shopping list on the Acme Inc. website. She attempts to access her account, but since she hasn't authenticated yet, she first needs to verify her identity.

Figure 12.6 The authorization code grant type. The user verifies their identity to access and work with data managed by the app's backend.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



Let's take the steps in figure 12.6 one by one and explain what happens there:

1. The user (Jeanny) wants to access data from the backend, so she asks the client (the app she's using) to display her shopping list. To retrieve the shopping list from the backend, the client needs to know who the user is

and prove to the backend that the user has verified her identity.

2. Since the user hasn't authenticated yet, the client redirects her to the IdP login page in a browser. The dotted arrows in the figure represent these redirects.
3. Facing the IdP's login page, Jeanny fills in her credentials and logs in. We'll assume she correctly filled in her username and password.
4. After Jeanny correctly enters her credentials, the IdP successfully authenticates her. The IdP then generates a unique, one-time-use code called an authorization code (hence the name of this grant type). The client can use this code to retrieve an access token (the "access card" it needs to prove to the resource server that Jeanny has authenticated). The IdP redirects back to the client providing the authorization code during the redirect (observe the dotted arrow in the diagram).
5. The client sends the authorization code back to the IdP and asks for the access token.
6. The IdP responds with an access token.
7. The client uses the access token to send requests to the backend (the resource server).
8. The backend response to the client's requests.

Great! The client receives the access token and uses it to send requests to the resource server. But how does the resource server know the token is authentic? How can it be sure that no one has tampered with or replaced the token entirely?

Let's revisit the concepts you learned in chapter 9. The most common approach today is to sign the token using a cryptographic public-private key pair. As you'll recall, the private key is used to sign the data, and only the entity authorized to sign it knows this private key (hence the term "private"). Then, anyone with the corresponding public key can validate that the data hasn't been tampered with.

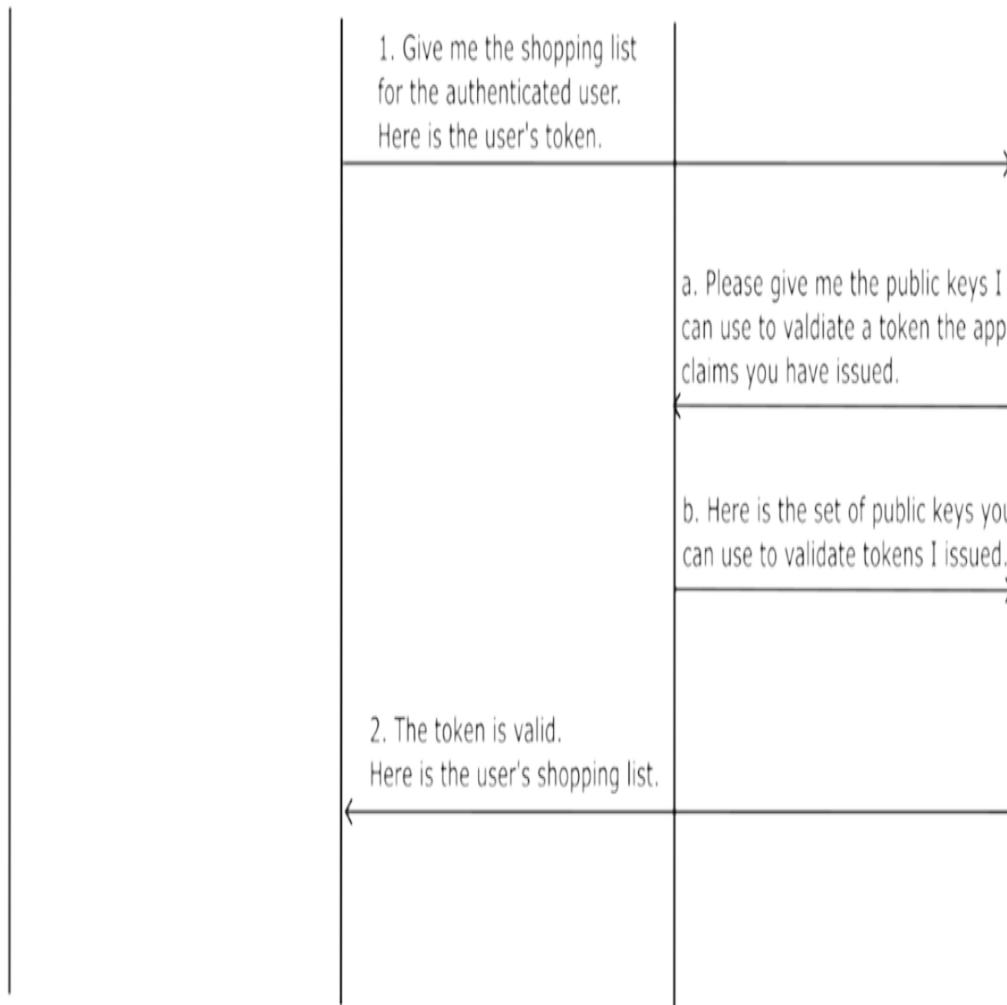
In this case, the IdP is the authorized entity responsible for generating access tokens. It signs the tokens it creates and makes its public keys available so that anyone can validate the tokens it has issued.

Figure 12.7 explains this process, which corresponds to steps 7 and 8 in the diagram from figure 12.6. The client sends a request to the resource server,

including an access token to prove that the user has authenticated. The resource server must verify that the access token is valid. To do this, it requests the public key from the IdP, which is used to validate the token's signature. The IdP provides the public key, and the resource server uses it to confirm the token's authenticity before responding to the client.

Figure 12.7 Verifying the authenticity of a signed token. When the resource server receives a signed token, it uses a public key to validate the token. The resource server obtains this public key from the IdP, allowing it to verify that the token is valid.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



You might have noticed that I used the plural form "keys" in the diagram. Typically, an IdP maintains multiple key pairs and randomly selects one each time it signs a token. This approach enhances security. If one of the private keys were to be compromised, it wouldn't affect all the access tokens the IdP has generated.

When the IdP generates a new access token, it also stores a unique identifier for the key pair used to sign the token. The resource server uses this identifier to determine which public key it needs to validate the token.

The implicit grant type legacy

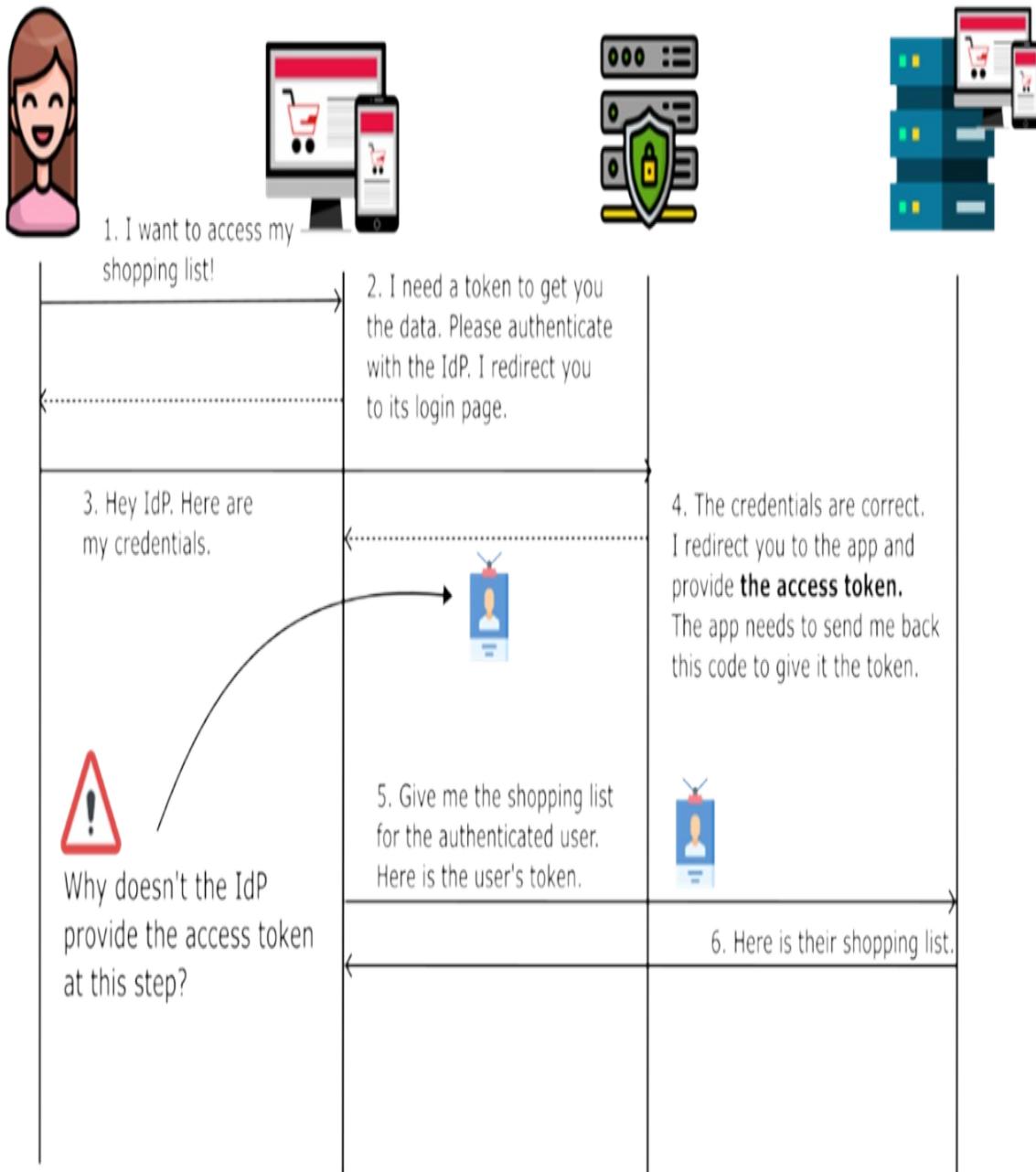
The authorization code grant type is challenging for almost everyone when they first encounter it. If you found it difficult and had to review the diagram and steps multiple times, congratulations—you’re human!

But like many others, once you start to understand the flow better, a question might come to mind: Why doesn’t the IdP provide the access token directly? Why complicate the process by sending an authorization code and then requiring it to be sent back in exchange for the actual access token?

The next figure illustrates the implicit grant type. Notice in step 4 where the redirect provides the access token directly, rather than an authorization code. Compare this diagram with Figure 12.6 to see the differences.

The implicit grant type. After the user authenticates, the IdP redirects back to the client, directly providing an access token. While this flow is simpler, it is more vulnerable to access token leakage.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



If you thought of this as an improvement in the flow, excellent—you've just discovered the implicit grant type! Unfortunately, we no longer use this grant type, as it's considered vulnerable and has been deprecated.

But what makes this flow vulnerable? And why does the additional

authorization code step solve the issues?

The key point to remember is that step 4 involves a redirect in the browser. Since it's a redirect, the token in the implicit flow is passed through the URL. Keep in mind that the token acts as an access card, and it should not be shared. There are several problematic aspects of this approach:

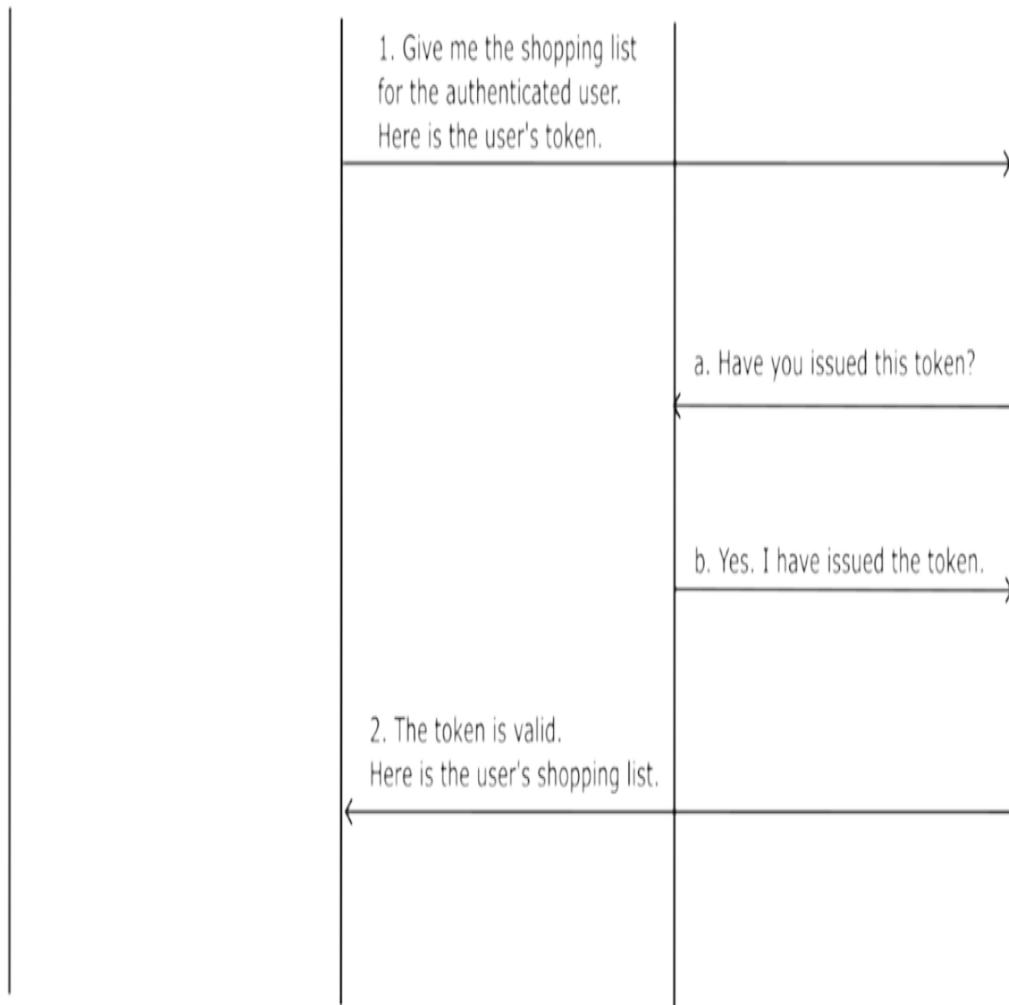
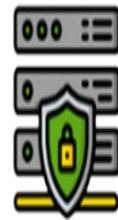
1. The URL can be easily forwarded to a different site.
2. The URL might be stored in the browser history making the token this way available also to browser plugins.
3. The URL might be exposed in logs together with the access token.

All these issues make the authentication flow vulnerable, a risk we refer to as "access token exposure." Our goal is to eliminate this vulnerability. By using the authorization code, the IdP sends the token back in an HTTP response, not through a redirect, which addresses the problems outlined in this sidebar.

A second approach to validate an access token is called "token introspection." In this method, the resource server directly queries the IdP to verify the token. The resource server sends the token to the IdP, asking whether it is valid. The IdP then responds by confirming or denying the token's validity. Figure 12.8 describes the introspection approach.

Figure 12.8 Token Introspection. When a token isn't cryptographically signed, the resource server must directly query the IdP to validate the token. This process is known as token introspection.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



While introspection may seem simpler from a logical standpoint—"just ask the entity that generated the token if it's valid"—it comes with a few disadvantages:

- The apps become highly coupled with the IdP – if the IdP is down the

apps cannot validate tokens.

- Frequent queries to the IdP put additional load on it, requiring more resources and potentially needing higher scaling.
- Apps exchange access tokens more frequently, increasing the risk of exposure.
- Since each validation requires a new HTTP request, introspection is a slow approach.

12.2.2 What actually are the tokens?

Our entire discussion in this chapter has focused on how to obtain access tokens—the essential "access cards" a client needs to access the resources exposed by the resource server. But in the end, what exactly are these tokens?

In this section, we'll dive into access tokens and cover everything you need to know about this crucial component in the OAuth 2 / OpenID Connect design. We'll first explore how apps use JSON Web Tokens (JWTs) as access tokens, but we'll also discuss scenarios where other token formats might be used.

We have covered JWTs in chapter 5, where we discussed the JSON Object Signing and Encryption (JOSE) standard. The next code snippet shall remind you about the shape of a signed JWT.

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IjBjN2E5NDdmLWFhYjAtNDFhNS04NWYwLWZiO
```

A JWT consists of three parts. The first two parts are JSON-formatted data that have been Base64 encoded. The third part is a cryptographic signature, which allows us to verify whether the token has been tampered with and is still valid.

The first part, known as the header, typically contains metadata about the token, such as the algorithm used for the cryptographic signature, the key ID, the token type, and more. The following code snippet shows an example of a JWT header in JSON format before it has been Base64 encoded.

```
{  
  "alg": "HS256",
```

```
"kid": "0c7a947f-aab0-41a5-85f0-fb898436e139",
"typ": "JWT"
}
```

The following code snippet shows the second part of the JWT, also known as the body or payload. This part typically stores data essential to the authentication process, such as the username of the authenticated user, the user's roles, the identifier of the client application that requested the token, the token's expiration time, and more.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

When the system uses JWTs as access tokens, the app's backend only needs to verify the token's signature and use the data within the token for authorization. This makes JWTs very convenient for this purpose. Access tokens typically expire after a short period, usually within an hour or less. This short lifespan makes it difficult for a stolen token to be misused.

In most cases, applications utilize tokens shaped as JWTs. While the OAuth 2 specification does not mandate a specific token format, the OpenID Connect protocol, which is built atop OAuth 2, mandates the use of JWTs. Consequently, as most implementations today depend on OpenID Connect, they also employ JWTs.

However, some applications that rely solely on the OAuth 2 specification may use tokens in various formats. Generally, tokens are classified as either opaque or non-opaque. Non-opaque tokens contain some form of readable information, while opaque tokens do not. JWTs are an example of non-opaque tokens, as they store data in their header and body. For opaque tokens the only way of validating them is using introspection (as discussed in figure 12.8).

12.2.3 The client credentials grant type

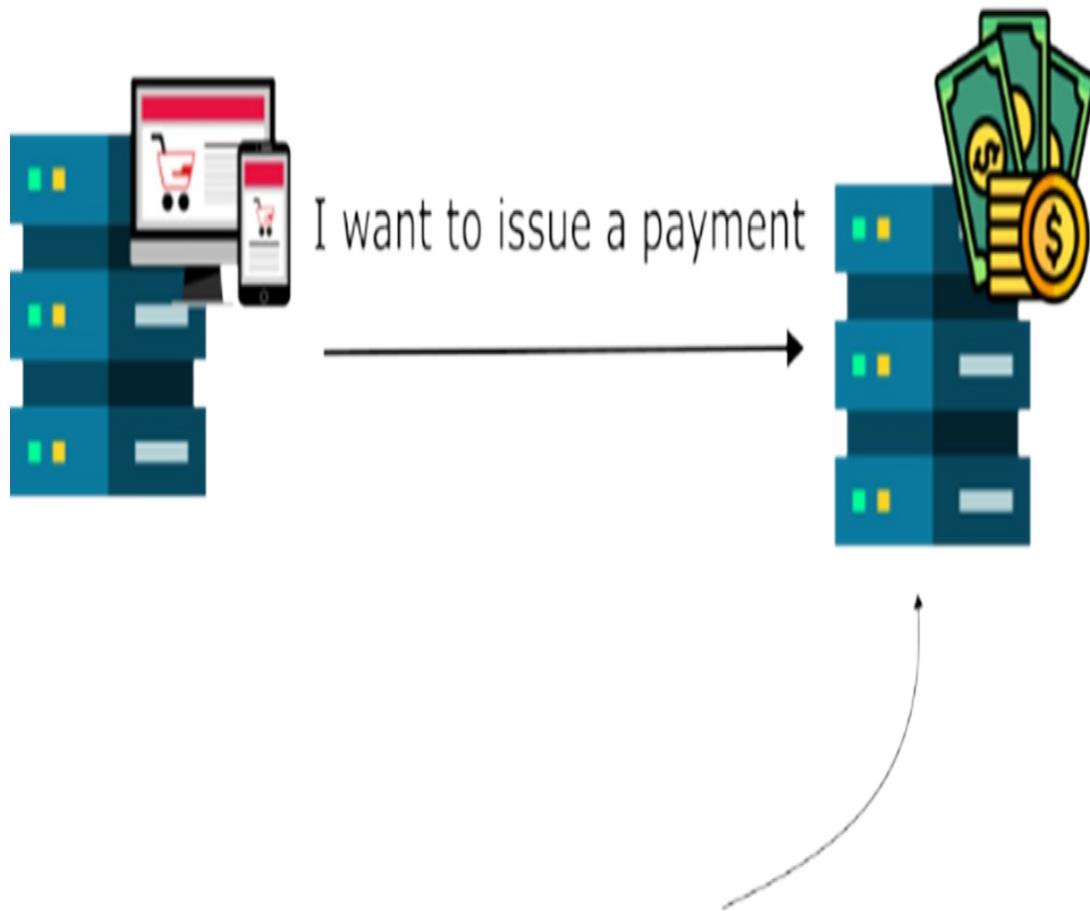
In real-world systems, applications often need to communicate with one another. In many instances, these applications must trust the messages they exchange. For this reason, authentication is also implemented when applications interact. OAuth 2 includes an authentication flow specifically for such scenarios. The client credentials grant type, which we discuss in this section (and also more in detail in chapter 16), provides a method for services to authenticate each other.

Unlike the authorization code grant type, the client credentials grant type does not involve a user. This grant type is used when one application needs to authenticate against another without user involvement, such as when two services exchange requests (figure 12.9).

Figure 12.9 Suppose Acme Inc. needs to issue payments. In this scenario, our app must communicate with a specialized service known as a payment service provider. This provider must verify that the requests are indeed coming from Acme Inc. and not another service. Additionally, it needs to ensure that the data has not been altered during transmission.

Acme Inc.

Payment Service Provider



How can the payment service provider ensure that the request came from Acme Inc?

You'll find the client credentials grant type significantly less complex than the authorization code grant type (figure 12.10). In this flow, the client, such as the Acme Inc. service, authenticates directly with the Identity Provider (IdP) using their client credentials. Upon successful authentication, the client receives an access token. This token can then be used to send requests to the resource server. In our example, the resource server is the Payment Service Provider that Acme Inc. needs to use to issue payments. The IdP in this scenario is an authorization server trusted by the Payment Service Provider.

Figure 12.10 Acme Inc. authenticates against an Identity Provider (IdP) and obtains a token. The

app then uses this token to send requests to another service, which authorizes them.

1. Acme Inc.



2. The identity provider (IdP)



3. The Payment Service Provider



1. Here are my credentials.
I need an access token to
send a payment request.

2. Credentials are correct.
Here is an access token.

3. I want to send
a payment request.

4. Your token is valid.
I completed your request.

12.2.4 Exercises

4. What are the four main actors in OAuth2 and OIDC?
5. What is the authorization code grant type?
6. Why not send the token directly in the redirect?
7. What is token introspection?
8. What is an access token?
9. What's the difference between opaque and non-opaque tokens?

12.3 Applying OpenID Connect to Acme Inc.

Let's apply the concepts from this chapter to our Acme Inc. example. We want to ensure that the endpoints our app exposes are protected and require user authentication through an IdP that we provide. For this, we'll use the Spring Authorization Server for the IdP implementation and Spring Security to manage endpoint authorization within our app.

While this book focuses more on security principles than on specific framework implementation details, we will not delve deeply into either the Spring Authorization Server or Spring Security. However, for those interested in learning more about these frameworks, an excellent starting point is *Spring Security in Action, Second Edition* by Laurențiu Spilcă (Manning, 2023).

Listing 12.1 below demonstrates the IdP configuration. We include a simplified IdP in our example to help understand the necessary configurations and to clarify its overall purpose. As shown in this concise YML file, the IdP requires at least:

- User credentials: The IdP authenticates the users, so it needs to manage their credentials.
- Client credentials: The IdP must be able to authenticate and differentiate between the applications it serves as clients. As discussed earlier in this chapter, an IdP might serve multiple apps. The IdP needs to recognize these apps and authenticate them, which is why the apps (clients) also need to have their own credentials.

Besides the client credentials, you can specify various configurations for each client. For example, this IdP configuration determines the grant types that the IdP allows for these clients, the redirect URIs that it accepts for redirections in the case of the authorization code grant type, and the scopes that the client may request.

Listing 12.1 The authorization code configuration

```
server:  
  port: 9000  
  
spring:  
  security:  
    user: #A  
      name: user #A  
      password: password #A  
  oauth2:  
    authorizationserver:  
      client: #B  
        oidc-client:  
          registration:  
            client-id: "oidc-client" #C  
            client-secret: "{noop}secret" #C  
            client-authentication-methods: #D  
              - "client_secret_basic" #D  
            authorization-grant-types: #E  
              - "authorization_code" #E  
            redirect-uris: #F  
              - http://127.0.0.1:8080/login/oauth2/code #F  
            scopes: #G  
              - "openid" #G
```

Let's begin setting up this Identity Provider (IdP) and conduct some tests. You can locate the IdP implementation in the acme_identity_provider project. We aim to personally execute the authorization code grant type to gain a deeper understanding of it. Here are the steps we will follow:

1. We'll access the OpenID Connect configuration URL to retrieve details about the IdP configuration. This information will include the endpoints that the IdP exposes for client app authorization and access token issuance.
2. We'll use a browser to simulate the client's redirection to the IdP, as

depicted in point 2 of figure 12.6, which outlines the authorization code grant type process.

3. We'll log in using user credentials to obtain an authorization code, corresponding to point 3 in figure 12.6.
4. We'll use cURL to send a request to the token endpoint to acquire an access token.

Let's start with step 1. Here's the Identity Provider's (IdP) OpenID Configuration URL. By default, the OpenID Configuration is accessible at the `/.well-known/openid-configuration` path. For our purposes, the server is hosted on localhost at port 9000. Therefore, the full URL is presented in the following snippet:

```
http://localhost:9000/.well-known/openid-configuration
```

You can enter the address in a web browser or use a tool like cURL to send a request. In the following snippet, you'll find a part of the response from the OpenID Connect configuration endpoint. I have truncated the response to highlight the essential details: the authorize endpoint and the token endpoint. These details are crucial for the subsequent steps.

```
{  
  "issuer": "http://localhost:9000",  
  "authorization_endpoint": "http://localhost:9000/oauth2/authoriz  
  "device_authorization_endpoint":  
  [CA]"http://localhost:9000/oauth2/device_authorization",  
  "token_endpoint": "http://localhost:9000/oauth2/token", #B  
  
  ...  
}
```

The next step is to use the authorization endpoint obtained from the OpenID Connect configuration to retrieve an authorization code. This simulates the process where the client application redirects the user to the IdP (Identity Provider) login page. During this step, the client application must include several essential parameters, which allow the IdP to understand the following:

- The grant type the client expects to use.
- The identity of the client requesting the authentication.
- The URL to which the IdP should redirect the user after a successful authentication.
- The scope for which the client sends the request.

The following snippet provides an example of the full request. The response_type parameter is set to code, indicating that the client requests the authorization code flow. The client identifies itself using the client_id parameter. The redirect_uri parameter specifies the URI where the client expects the IdP to redirect the user after a successful authentication. The scope parameter defines the requested scope, which in this case is openid.

This request ensures the client can properly initiate the authentication flow with the required parameters.

```
http://localhost:9000/oauth2/authorize?response_type=code&client_
```

After the user successfully authenticates, the IdP redirects the user to the URI specified by the client. Along with the redirection, the IdP provides a unique authorization code that can only be used once. The following snippet illustrates an example of the URI to which the IdP will redirect the user.

This code will then be exchanged for an access token in the next step of the authorization process.

```
http://127.0.0.1:8080/login/oauth2/code?code=d8jhmY2i1...
```

In the next step, the app uses the authorization code provided by the IdP to obtain an access token. This step can also be simulated. To act as the client app and retrieve an access token, a request must be sent to the access token endpoint. Similar to how we retrieved the authorization endpoint, the access token endpoint is obtained from the IdP's OpenID Connect configuration.

To obtain an access token using the token endpoint, we need to send a POST request with the following parameters:

- The grant_type parameter is set to authorization_code, which informs the IdP that we are using the authorization code grant type. This allows the IdP to expect an authorization code value in the request.
- The code parameter holds the authorization code sent by the IdP in the previous step when redirecting the user back to the client.
- The redirect_uri contains the value of the URI where the IdP sent the authorization code during the redirect.
- The client_id identifies the client requesting the access token. This value must match the client for which the authorization code was initially issued.

By providing these parameters, the client can successfully exchange the authorization code for an access token. The next snippet demonstrates the access token request. You can use cURL or a similar tool to send this request. Notice that the request employs HTTP Basic authentication, using the client credentials (client ID and secret) for authentication by default.

Please note that in this example, the authorization code value is truncated for brevity. However, when sending your request, be sure to include the full authorization code value.

```
curl --location 'http://localhost:9000/oauth2/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'Authorization: Basic b2lkYy1jbGllbnQ6c2VjcmV0' \
--data-urlencode 'grant_type=authorization_code' \
--data-urlencode 'code= d8jhmY2i1...' \
--data-urlencode 'redirect_uri=http://127.0.0.1:8080/login/oauth2'
--data-urlencode 'client_id=oidc-client'
```

The following snippet shows the response to the access token request. The tokens you receive will always be in JWT format when using the OpenID Connect protocol. Additionally, with OpenID Connect, you'll receive two distinct tokens:

- An access token, which the app uses for authorization purposes.
- An ID token, which contains details about the authenticated user that the resource server may need.

These tokens play separate but complementary roles in managing

authentication and authorization within the OpenID Connect protocol.

```
{  
    "access_token": "eyJraWQiOiJiMzIyODUzNS...",  
    "scope": "openid",  
    "id_token": "eyJraWQiOiJiMzIyODU...",  
    "token_type": "Bearer",  
    "expires_in": 299  
}
```

In the previous snippet, I truncated the tokens for brevity. However, in the next listing, you'll find the Base64-decoded version of the access token. Notice that the header includes the key ID (kid), which the resource server uses to identify the key required to validate the token. Additionally, the token's body contains important details about both the user and the client. This decoded information provides transparency into how the token is structured and the data it carries for the authorization process.

Listing 12.2 The Base64 decoded access token

```
{  #A  
  "kid": "b3228535-0e94-4c8a-8ee9-38cded00323a",  #B  
  "alg": "RS256"  
}  
  
{  #C  
  "sub": "user",  #D  
  "aud": "oidc-client",  #E  
  "nbf": 1727537173,  
  "scope": [  
    "openid"  
,  
  "iss": "http://localhost:9000",  
  "exp": 1727537473,  
  "iat": 1727537173,  
  "jti": "15e1912f-57e6-4974-8660-7e240076bc7b"  
}
```

12.3.1 Answers to exercises

1. What is Single Sign-On (SSO)?
It allows users to log in once and access multiple apps without logging in again. One login, many apps.
2. What is an Identity Provider (IdP)?
It's a special app that handles authentication for other apps. It checks who you are and issues a token as proof.
3. Why do we separate authentication into its own app?
It makes things more secure, easier to manage, and allows many apps to reuse the same login logic.
4. What are the four main actors in OAuth2 and OIDC?
User (you), Client (the app), Identity Provider (IdP), and Resource Server (backend that serves data).
5. What is the authorization code grant type?
It's a flow where the client gets an access token through a secure exchange involving an authorization code and redirects.
6. Why not send the token directly in the redirect?
That's unsafe. It could leak in the browser history or logs. The extra step with the code avoids this.
7. What is token introspection?
It's when the backend checks with the IdP to see if a token is still valid.
It's slower but useful when you can't use JWTs.
8. What is an access token?
It's a digital pass that lets an app talk to a backend on your behalf. It proves you're logged in.
9. What's the difference between opaque and non-opaque tokens?
Opaque tokens don't contain data; Non-opaque tokens are readable and can be validated without asking the IdP.

12.4 Summary

- Single Sign-On (SSO) simplifies user authentication by allowing employees and customers to use a single set of credentials for multiple applications.
- OAuth2 and OpenID Connect are protocols used to implement SSO by separating authentication responsibilities into an Identity Provider (IdP).
- Among the authentication flows the OAuth 2 specification describes the most used ones are:

- The authorization code grant type is the most common, used when user interaction is required.
 - The client credentials grant type is used when one application needs to authenticate another without user involvement.
- Using OAuth 2/OpenID Connect implies multiple actors in the authentication flow:
 - The user (e.g., employee or customer)
 - The client (app or service used)
 - The IdP (also known as authorization server) – a service offering authentication capabilities to multiple clients.
 - The resource server - app backend that handles data and authorization
- Tokens are central to OAuth2 and OpenID Connect:
 - Access tokens which are used for authorization
 - ID tokens which contain user information
- JWTs (JSON Web Tokens) are commonly used for token formatting and include a header (metadata), a payload (user/client details), and a signature for validation.
- Token validation involves cryptographic methods or token introspection, ensuring secure communication between applications.

13 Deepening security with OpenID Connect

This chapter covers

- Boosting security with Proof Key for Code Exchange (PKCE) in the authorization code flow.
- Simplifying user logins and maintaining sessions with refresh tokens.
- Enhancing user identity management with OpenID Connect's identity layer and additional features.
- Implementing multitenancy with OIDC

As Acme Inc. grew rapidly, we realized we needed a better and safer way for users to log in. That's when we decided to set up Single Sign-On (SSO), which lets users access multiple apps with just one login. To do this right, we had to separate the job of checking who a user is (authentication) from the app that handles the user's data (the backend). This setup makes things safer and easier to manage.

Of course, we couldn't just make it up as we went along—we needed a solid, trusted system to guide us. That's how we found OAuth 2 and OpenID Connect (OIDC). OAuth 2 is a framework that lets apps get permission to access certain user data without needing passwords. OpenID Connect builds on OAuth 2 by adding a way to confirm who the user is. It's like OAuth 2 is the pizza crust, and OpenID Connect is the delicious cheese and toppings. Who wants plain crust anyway?

In chapter 12, we talked about how OAuth 2 and OIDC work. We covered how tokens—like access tokens and ID tokens—help apps talk to each other securely. We also explained the different ways (called grant types) that apps can ask for these tokens. Knowing these steps is key to setting up secure and smooth logins.

In this chapter, we'll dive deeper into how Acme Inc. can use advanced tools

like Proof Key for Code Exchange (PKCE) to make the authorization process more secure, especially for public clients. We'll also explore how refresh tokens can keep users logged in without asking them to sign in again and again. Plus, we'll unpack how OpenID Connect adds an identity layer on top of OAuth 2, providing even more benefits like user profile information and stronger security measures. With these tools in place, Acme Inc. can confidently scale its systems while keeping user accounts safe and the login experience smooth.

13.1 Augmenting the authorization code grant type with PKCE

A client must first obtain a valid access token to access resources protected by a backend application. This token is acquired through a specific authorization flow known as a grant type. In chapter 12, we explored one of the most widely used grant types - the Authorization Code flow. This grant type is particularly suitable when the client needs to access resources on behalf of a user, providing an added layer of security by involving user authentication. Figure 13.1 provides a visual reminder of how this authorization flow operates.

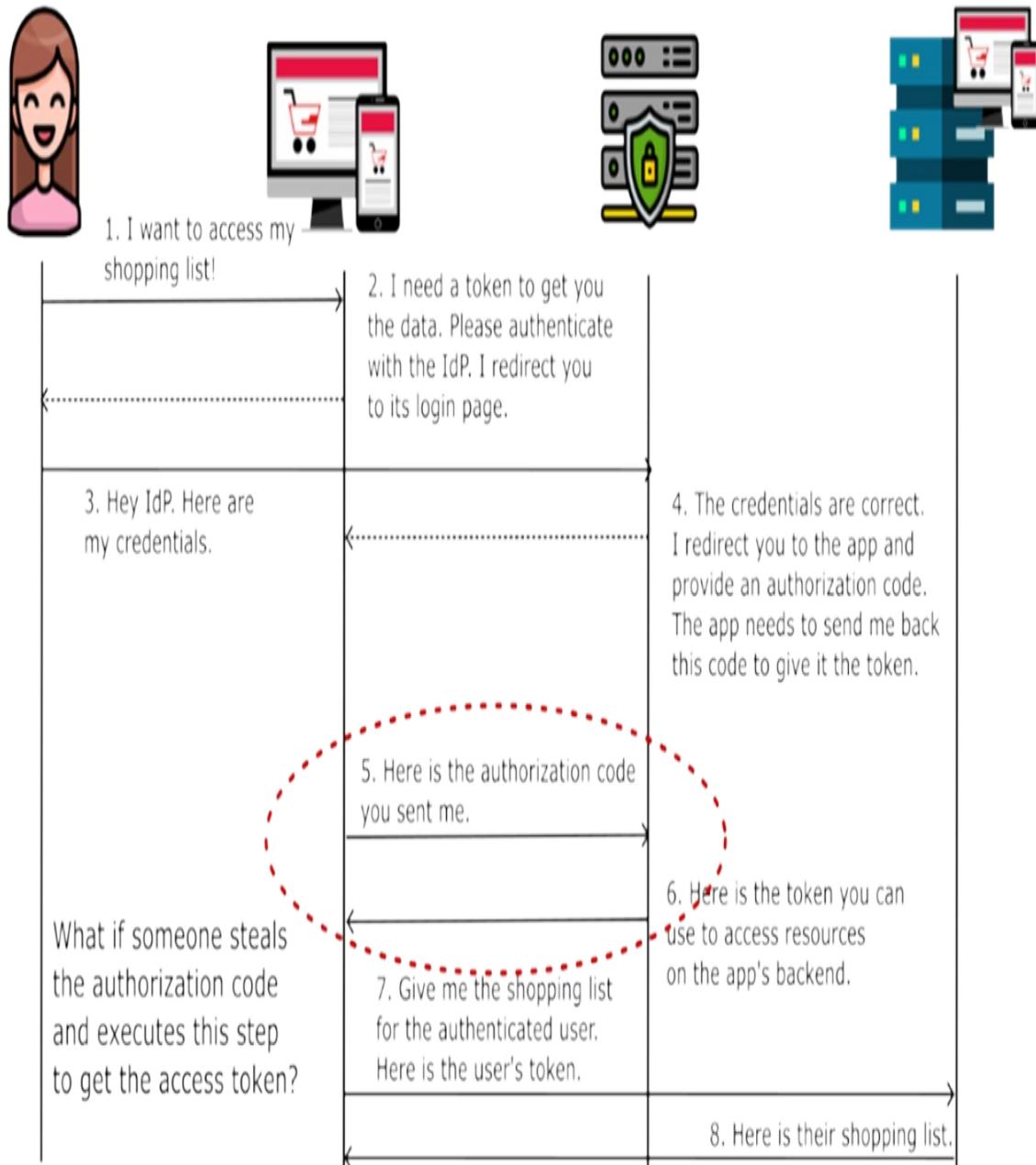
Let's revisit the steps illustrated in figure 13.1. Jeanny, our user, wants to view her shopping list in the Acme Inc (step 1). shopping app. Since Jeanny's shopping list contains private information that belongs exclusively to her, the client app must obtain explicit permission to access this data on her behalf. To achieve this, the client redirects Jeanny to authenticate with a trusted authorization server, ensuring only authorized access to her personal information is granted (step 2).

Jeanny enters her credentials to authenticate with the authorization server (step 3). Upon successful authentication, the authorization server redirects her back to the client application, delivering a unique authorization code (step 4). The client app then uses this authorization code to securely request and obtain an access token (steps 5 and 6), which grants it permission to access Jeanny's protected data (steps 7 and 8).

Figure 13.1 The Authorization Code Grant Flow. The client application requests access to

resources protected by a backend application (resource server) on behalf of a user. To initiate this process, the client redirects the user to authenticate with a trusted authorization server. Upon successful authentication, the client receives an authorization code, which it then exchanges for an access token. With a valid access token, the client can securely access the protected resources hosted by the backend application.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



Alright! You might wonder, “What if someone somehow gets their hands on the authorization code?” Fair question! While it’s undoubtedly tricky, we can’t pretend it’s impossible—nothing’s 100% foolproof. Typically, since the authorization code is sent directly from the client to the authorization server, the request must also be authenticated using the client’s credentials. These credentials are essentially a username and password that the client app uses to prove its identity to the authorization server.

So, for an attacker to succeed, they’d need to pull off a double heist: intercept the authorization code and possess the client credentials to request the access token. Sounds like a tall order, right? Well, here’s the catch—client credentials aren’t exactly guarded like crown jewels. They’re stored within the client app, making them vulnerable if the app isn’t well-secured (which is likely the case of a public client). So while this scenario is tough to pull off, it’s not mission impossible. (Cue the suspenseful music!)

But this is where PKCE (Proof Key for Code Exchange) comes to the rescue! With just a small tweak to the authorization flow, we can significantly strengthen security. Let’s take a moment to carefully analyze the root of the problem. The vulnerability arises between steps 4 and 5 in the flow (refer back to figure 13.1).

After the user successfully authenticates in step 4, an attacker could potentially intercept the authorization code and use it to impersonate the legitimate client. This would allow the attacker to gain unauthorized access to the user’s sensitive resources. And let’s be honest—Jeanny definitely wouldn’t be thrilled about someone snooping through her private shopping list!

Figure 13.2 illustrates the enhancements made to the Authorization Code flow. To strengthen security, the client performs an additional task before redirecting the user for authentication: it generates a data pair known as the code challenge and the code verifier. This process is straightforward and involves two key steps:

1. *Generate the Code Verifier:* The client creates a random piece of data—typically a byte array—that is then Base64-encoded for easier representation and transmission. This random value is called the code

verifier. At its core, it's simply a unique, unpredictable string.

2. *Create the Code Challenge*: The client applies a cryptographic hash function (commonly SHA-256) to the code verifier to produce the code challenge. Need a quick refresher on how hash functions work? We covered them in chapter 4, so feel free to revisit that section!

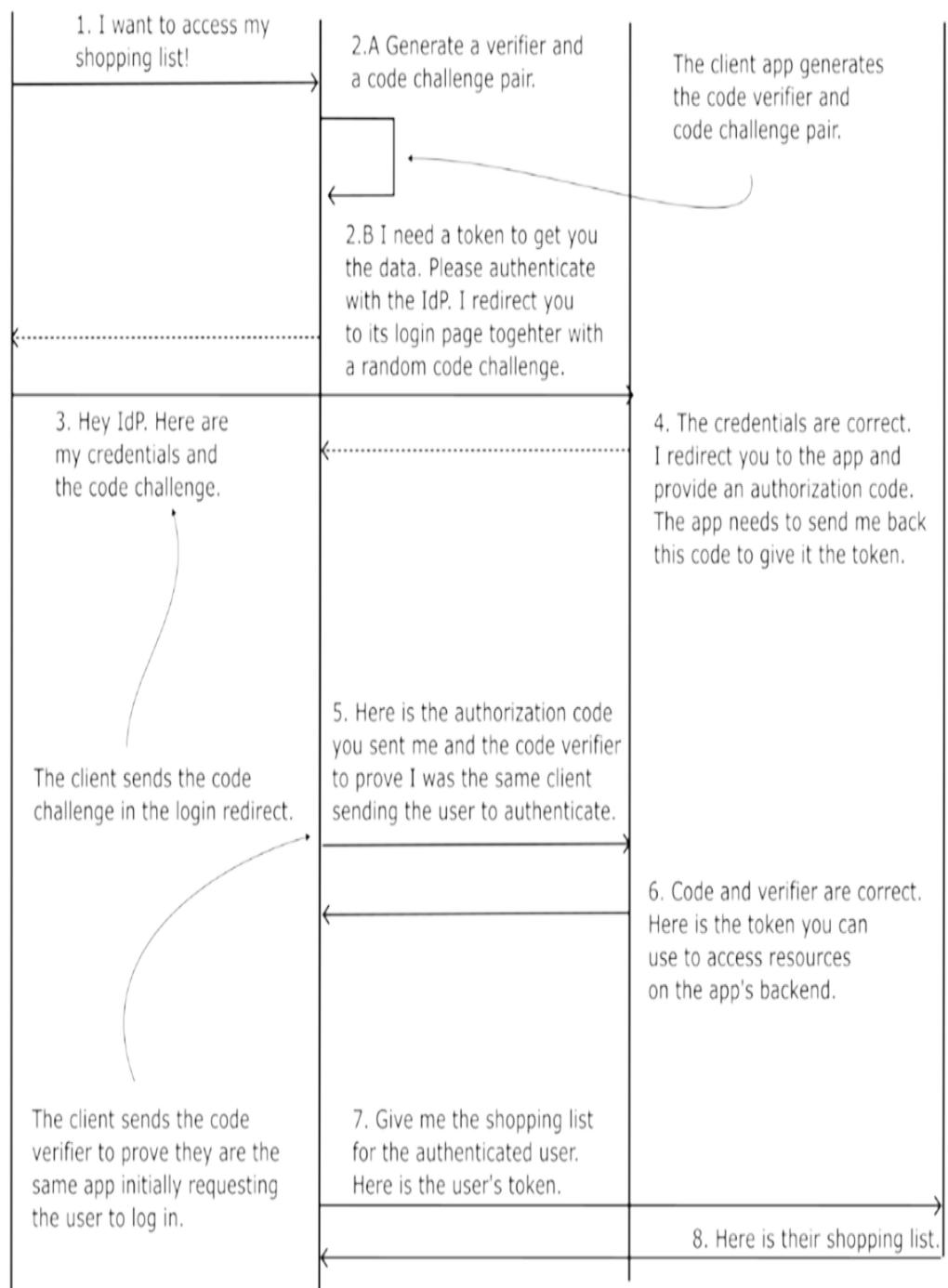
In figure 13.2, the generation of the code challenge and code verifier is depicted as step 2.A. The crucial part comes: the code challenge (the hashed version of the verifier) is sent to the authorization server as part of the user's login redirect (step 3).

The authorization server links the authentication session to that specific code challenge when the user successfully logs in. This association ensures that only the client possessing the correct code verifier can later exchange the authorization code for an access token.

When the client application requests an access token, it must now provide the authorization code and the correct code verifier (step 5). Without the valid code verifier, the authorization server will refuse to issue an access token, effectively blocking unauthorized access.

Figure 13.2 Proof Key for Code Exchange (PKCE) Flow. Before initiating the authentication process, the client generates a unique and random pair of values: a code challenge and a code verifier. The client uses this pair throughout the authorization flow to enhance security, ensuring no unauthorized party can intercept or misuse the authentication process to impersonate the user.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



It's important to remember that the code verifier is never transmitted over the network. Only the client application knows this secret value, making it impossible for attackers to intercept or reuse it. Additionally, the verifier is valid only for the current session. Each new authorization flow generates a fresh, random verifier and challenge, ensuring that every session is uniquely protected.

Can the code verifier be guessed using the code challenge (sent during the login redirect)? The answer is a solid no.

The code challenge is generated using a cryptographic hash function applied to the code verifier. Cryptographic hash functions are designed to be one-way functions, meaning it's practically impossible to reverse the process and retrieve the original input from the output. In this case, even if an attacker intercepts the code challenge, they cannot derive the code verifier from it.

This property of the hash function ensures that the code challenge alone is useless to any hacker attempting to exploit the flow. The verifier remains secret and secure, protecting the authorization process from interception and tampering (figure 13.3).

Figure 13.3 Relationship Between Code Verifier and Code Challenge. The code verifier is a securely generated random value. The code challenge is derived by applying a cryptographic hash function (commonly SHA-256) to the code verifier. This one-way transformation strengthens the security of the authorization process by ensuring the verifier cannot be reverse-engineered from the challenge.



CODE VERIFIER

cd81f8bddf20e42376528de6b67b0a7f



CODE CHALLENGE

aa3ce5d8f2023d4ccb72932d2b223b06b
56f6957de61d91eb520b75cf47f6609

PKCE, as an enhancement to the Authorization Code flow, is highly effective in many scenarios—particularly for public clients, third-party integrations, browser-based applications, and even IoT devices. These environments often lack secure storage for client secrets, making them more vulnerable to attacks like authorization code interception.

However, like any security measure, PKCE can become unnecessary or even overengineered if used in contexts with little to no added value. In private or highly controlled environments, where the risk of attacks on the authorization code flow is minimal, implementing PKCE may not justify the extra complexity.

Additionally, PKCE is specifically designed to prevent user impersonation by securing user-driven authorization flows. In machine-to-machine (M2M) communication—where no user is involved and systems securely exchange credentials—PKCE offers little benefit. In such cases, using the Client Credentials flow with properly managed secrets is more appropriate.

Let's wrap up this section with some key best practices—because security is no place for shortcuts!

- Always use PKCE with public clients (like mobile apps, single-page apps, and IoT devices) to prevent authorization code interception. Without it, you're practically leaving the front door wide open!
- Never disable PKCE where it's supported—seriously, don't do it. OAuth 2.1 now requires PKCE for all authorization code flows, so skipping it is like driving without a seatbelt.
- Use SHA-256 for the code challenge instead of the plain method (S256 over plain) for stronger security. Hashing with SHA-256 is like using a steel vault, while the plain method is more like hiding your keys under the doormat.

Remember always to follow best practices and keep your apps locked down tighter than a jar of pickles!

13.1.1 Exercises

1. What is PKCE and why is it important?
2. How does PKCE work?
3. Why is the code challenge secure?
4. When should you use PKCE?

13.2 Using refresh tokens to simplify authentication

Tokens are like keys to the kingdom—unlocking access to resources protected by the backend. However, as we discussed in chapter 12, leaving those keys valid forever isn’t exactly the brightest idea. Imagine giving someone a house key that never expires... not great, right? That’s why access tokens are designed to expire after a short period, typically within minutes, to minimize security risks.

But here’s the catch: do we really want to force users to log in every 20 minutes? Put yourself in their shoes—after the third login prompt, you’d probably be ready to toss your phone out the window.

So, how do we strike a balance between keeping tokens secure and keeping users happy? The answer is simple (and much friendlier): refresh tokens. In this section, we’ll explore how refresh tokens let us maintain security without driving users up the wall.

Figure 13.4 visually illustrates the Refresh Token flow. To use this flow, a client must first register with the authorization server to receive refresh tokens. Once registered, the client will receive a refresh token alongside every access token it obtains.

If you refer back to chapter 12, where we demonstrated how to send a request to the /token endpoint to obtain an access token, you might recall the structure of the response. The response we received closely resembles the example shown in the following snippet.

```
{  
  "access_token": "eyJraWQiOiJiMzIyODUzNS...",  
  "scope": "openid",  
  "token_type": "Bearer",  
  "expires_in": 299  
}
```

If the client is registered to receive refresh tokens, the response will be slightly different, as shown in the following snippet. In addition to the access token, the response will now include a refresh token, enabling the client to obtain new access tokens without requiring further user authentication.

```
{  
  "access_token": "eyJraWQiOiJiMzIyODUzNS...",  
  "refresh_token": "eaJqaEEi0iKYMzIyODUzPL...", #A  
  "scope": "openid",  
  "token_type": "Bearer",  
  "expires_in": 299  
}
```

This means that when a user authenticates for the first time using the Authorization Code flow (with or without PKCE), the client doesn't just receive an access token—it also gets a refresh token. The refresh token allows the client to request new access tokens without requiring the user to log in again, keeping the experience smooth.

As shown in figure 13.4, Jeanny is back in the Acme Inc. app, ready to check out her carefully curated shopping list. She's already authenticated, spent some time browsing through products, and added some fantastic items to her cart—some of them even 40% off! (Who can resist a good sale?) Now, she's ready to review her list and proceed to checkout.

But wait! When the client app tries to use the access token, the authorization server rejects it—because it's expired. Clearly, Jeanny was on a serious shopping spree!

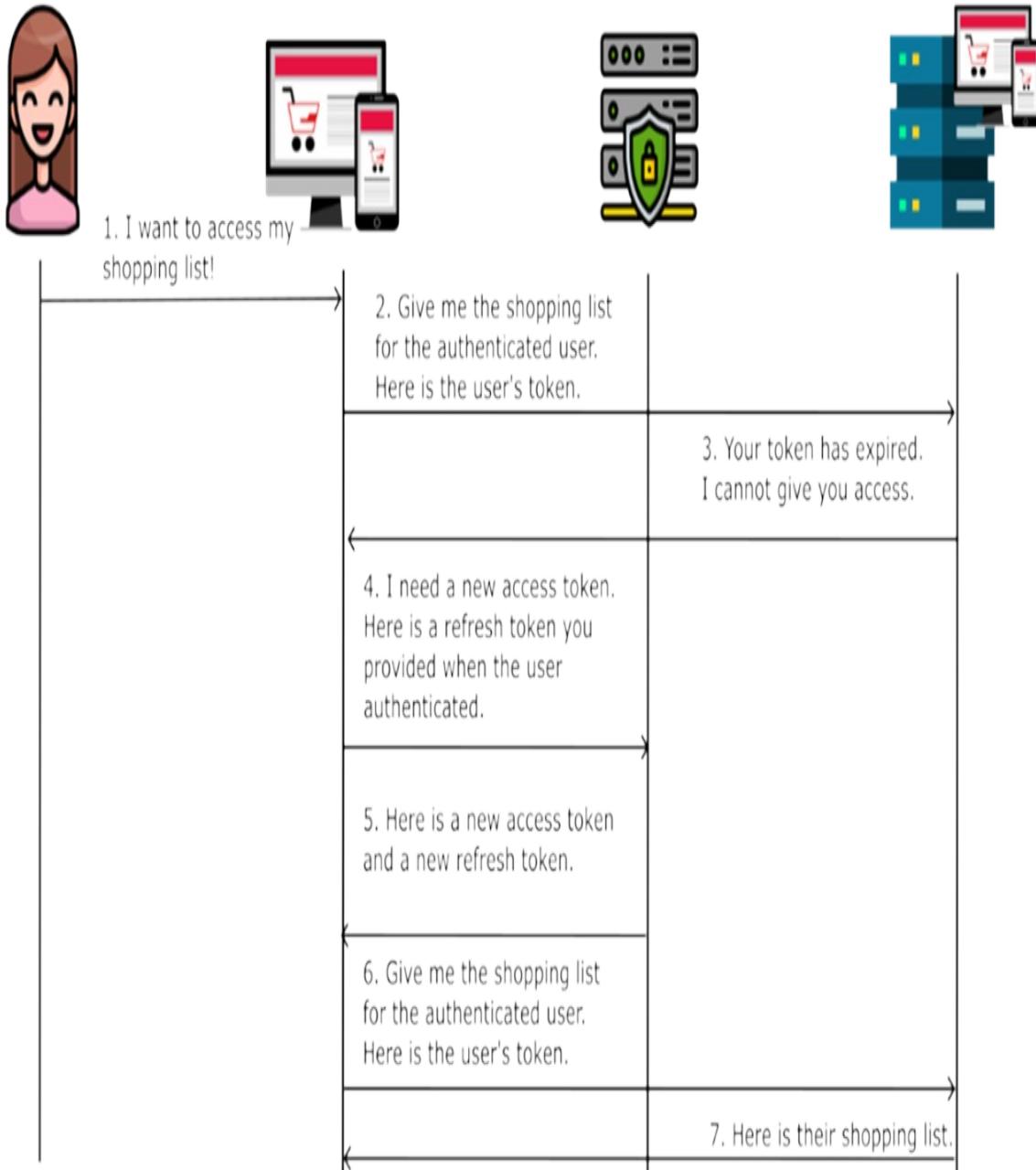
Now, asking Jeanny to log in again would totally kill the vibe. No one wants to type passwords when they're in checkout mode! So, instead of disrupting her smooth shopping experience, the client app cleverly uses the refresh token it received during Jeanny's first authentication.

The app sends a request to the /token endpoint, this time using the refresh token to ask for a new access token. Since the refresh token is still valid, the authorization server responds with a brand-new access token and a shiny new refresh token.

With the new access token in hand, the client app seamlessly retrieves Jeanny's shopping list—no password prompts, no interruptions. Jeanny stays happy, and her discount deals stay safe.

Figure 13.4 The Refresh Token flow. When a client is registered to use refresh tokens, it receives both an access token and a refresh token from the authorization server. The client can later use the refresh token to seamlessly request a new access token without requiring any additional user interaction. This approach maintains security while providing a smoother, uninterrupted user experience.

1. The user (Jeanny)
2. The app Jeanny uses in a web browser or mobile.
3. The identity provider (IdP)
4. Jeanny's app backend (resource server)



Okay! I know what you're thinking: Does this approach weaken security a little too much? What if someone steals a refresh token? Could they generate new access tokens endlessly and gain indefinite access?

While refresh tokens offer significant advantages (as discussed in this

section), handling them with care is crucial. Losing one could lead to serious security risks because refresh tokens are typically longer-lived than access tokens.

But don't panic! Both OAuth 2.0 and OpenID Connect provide robust mechanisms to secure refresh tokens and minimize potential threats. These include secure storage, token binding, refresh token rotation, and prompt revocation strategies, designed to make stealing and misusing refresh tokens difficult.

First, we need to consider the type of client we're securing. If it's a confidential client—like an internal Acme Inc. app used exclusively by the company's employees—the app can securely store refresh tokens because it typically operates in a controlled and secure environment (e.g., backend servers or corporate networks). In this case, securely managing tokens is much more straightforward.

However, extra precautions are necessary if the client is a public application—such as the Acme Inc. shopping app that Jeanny uses. Public clients (like mobile apps or single-page web apps) are more vulnerable because they can't securely store secrets.

In these situations, it's strongly recommended that refresh tokens be paired with PKCE (Proof Key for Code Exchange). This combination adds a critical layer of security by ensuring that only the legitimate client who obtained the refresh token can use it to request new access tokens.

Another crucial point to emphasize is that all communication between the client app, the authorization server, and the resource server must always occur over a secure transmission using TLS/SSL (see chapter 11 for a refresher). This encryption ensures that sensitive data—like access tokens, refresh tokens, and user credentials—cannot be intercepted or tampered with during transmission. Think of it as adding another solid brick to the security wall, making it even harder for attackers to find a way in.

Here are a few additional best practices for managing refresh tokens to help keep your app secure and resilient against attacks:

- Minimize Scopes: Issue refresh tokens with limited scopes to restrict the level of access they provide. This way, even if a refresh token is compromised, the damage is contained to specific, less-sensitive resources.
- Implement Token Rotation: Enable refresh token rotation, where a new refresh token is issued every time the old one is used—and the previous token is immediately invalidated. This makes it much harder for attackers to reuse stolen tokens since they become useless after a single use.
- Set Expiration and Enable Revocation: Configure refresh tokens with expiration times and allow the authorization server to revoke them if suspicious activity is detected. This reduces the risk of long-term misuse and gives you better control over token lifecycles.

13.2.1 Exercises

5. What are refresh tokens for?
6. Why not make access tokens last longer?
7. How do you protect refresh tokens?

13.3 Supporting identity management with OpenID Connect

We've explored the fundamentals of how OAuth 2.0 and OpenID Connect (OIDC) function. Now, let's dive into why OIDC decided to crash the OAuth 2.0 party—and brought its own snacks to make things better. The most important of them being:

- *The Identity Layer and ID Token* - OIDC adds an identity layer with the ID Token, providing a secure and standardized way to authenticate users.
- *The UserInfo Endpoint*—OIDC introduces the UserInfo Endpoint, which allows applications to easily retrieve additional user profile information.
- *Protection Against Replay Attacks and CSRF* - OIDC strengthens security by using parameters like nonce and state to prevent replay

- attacks and Cross-Site Request Forgery (CSRF).
- *Session management and logout* - OIDC standardizes session management and logout processes, enabling seamless user sign-in and sign-out experiences across applications.

13.3.1 The Identity Layer and ID Token

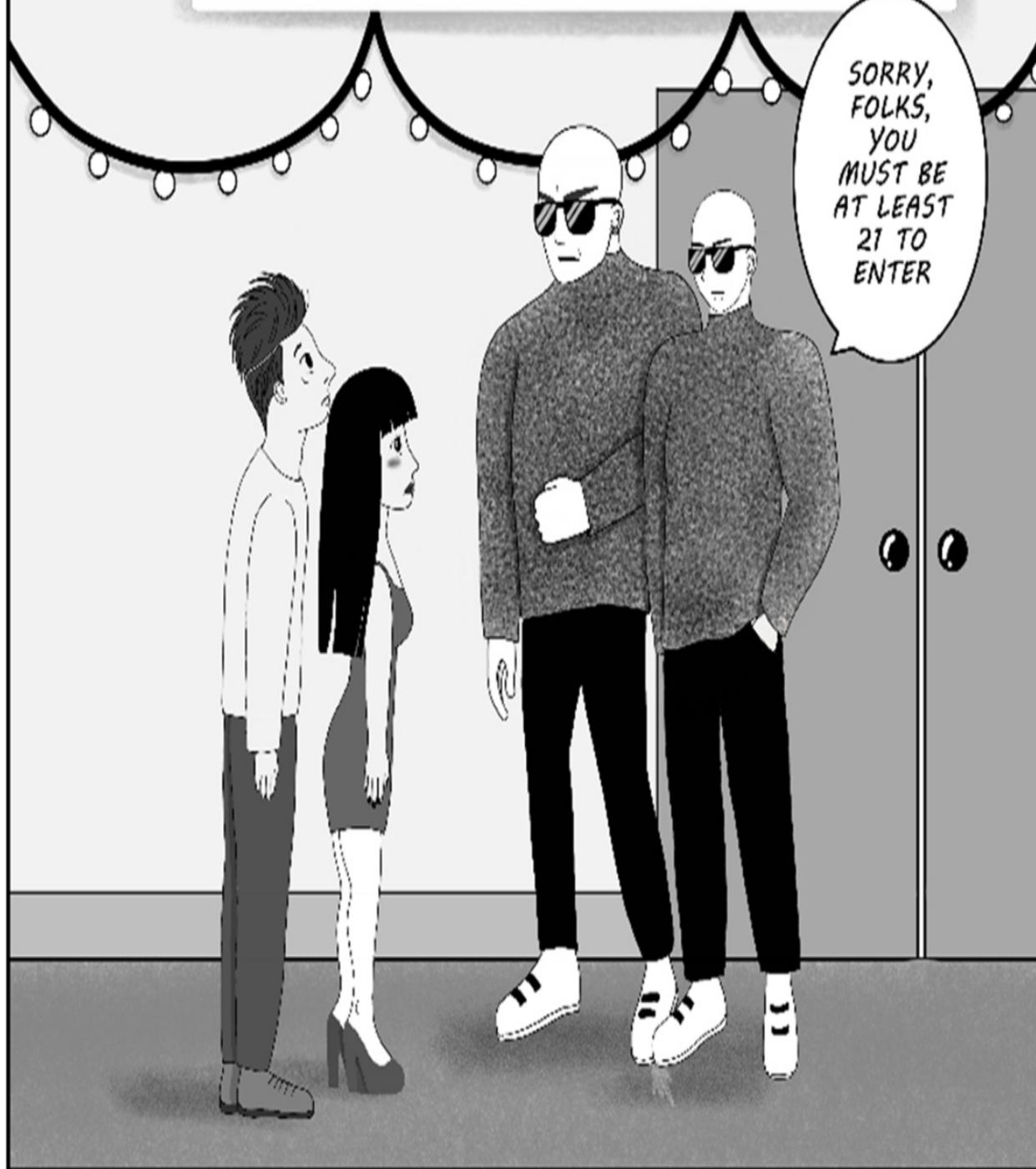
OAuth 2.0 is a fantastic authorization framework. It hands out access tokens like VIP passes, letting apps access user data without exposing passwords. It's great for saying, "Yes, this app can look at your photos," but not so great at answering the question, "Who are you, really?"

That's where OpenID Connect steps in. Think of OAuth 2.0 as a bouncer who checks if you're allowed into the club (figure 13.5). OIDC is the ID scanner that says, "Ah, you're Alex, born on July 5th, and here's your loyalty card."

Figure 13.5 OAuth 2.0 is like a bouncer at a club—it checks if you're allowed to enter but has no idea you're a VIP member.

BEST CLUB IN TOWN

SORRY,
FOLKS,
YOU
MUST BE
AT LEAST
21 TO
ENTER



OAuth 2.0 wasn't built for authentication—it's all about authorization. Sure, developers tried bending it to handle authentication, but it was like using a fork to eat soup. It kind of worked but spilled a lot of data in the process. OpenID Connect was created to fill this gap, officially adding a secure, standardized way to verify who a user is.

So, why was OpenID Connect needed?

Because OAuth 2.0 was never meant to answer the question, “Who’s there?” It only handled “What can they do?” OIDC gives applications both the keys to the door and the guest list, making authentication smoother, safer, and much less awkward.

Technically speaking, OpenID Connect (OIDC) enhances OAuth 2.0 by introducing an additional token alongside the access token. This new token is called the ID Token, and its primary purpose is authentication. While the access token grants permission to access resources, the ID Token provides a standardized way to identify who requests access.

The ID Token contains user-specific details—such as the user’s ID, email, and other profile information—offering a reliable method for verifying the requester’s identity.

In the following snippet, you’ll see the ID Token included in the response. The authorization server issues this token when the client sends a request to the /token endpoint. However, the ID Token is only present if the authorization server supports and enables the OpenID Connect protocol. This is because the ID Token is exclusive to OIDC and isn’t part of the standard OAuth 2.0 specification.

In short, OAuth 2.0 says, “Here’s your access pass,” and OIDC adds, “By the way, here’s proof of who you are.”

```
{  
  "access_token": "eyJraWQiOiJiMzIyODUzNS...",  
  "refresh_token": "eaJqaEEi0iKYMzIyODUzPL...",  
  "scope": "openid",  
  "id_token": "eyJraWQiOiJiMzIyODU..." , #A
```

```
        "token_type": "Bearer",
        "expires_in": 299
    }
```

Speaking of tokens... Remember back in chapter 12 when we talked about how access tokens can come in different formats? These days, the most commonly used format is the JSON Web Token (JWT) because it's compact, self-contained, and easy to verify. However, in certain scenarios, you might need to use opaque tokens—those mysterious tokens that contain no readable information (as we discussed in chapter 12). Opaque tokens require the resource server to check with the authorization server to understand what the token represents.

But here's an important distinction: the ID Token in OpenID Connect (OIDC) is always a JWT. Why? Because its main purpose is to carry information about the user—like their identity and authentication details. To fulfill this role, the ID Token must be non-opaque, meaning it has to contain readable and verifiable claims. And today, the most efficient and standardized format for securely packaging this kind of information is the JWT.

13.3.2 The UserInfo Endpoint

Another key enhancement that OpenID Connect (OIDC) brings to the table over OAuth 2.0 is the introduction of a standardized endpoint for retrieving user details - the *Userinfo Endpoint*.

This endpoint lets client applications securely fetch additional user profile information (like name, email, and profile picture) directly from the authorization server. It complements the ID Token by providing more dynamic or extensive user data that might not be included in the token itself.

The *Userinfo Endpoint* is vital to the identity layer that OIDC adds to OAuth 2.0. It works hand-in-hand with the ID Token and OIDC's authentication mechanisms to provide a complete and secure identity solution.

13.3.3 Protection Against Replay Attacks and CSRF

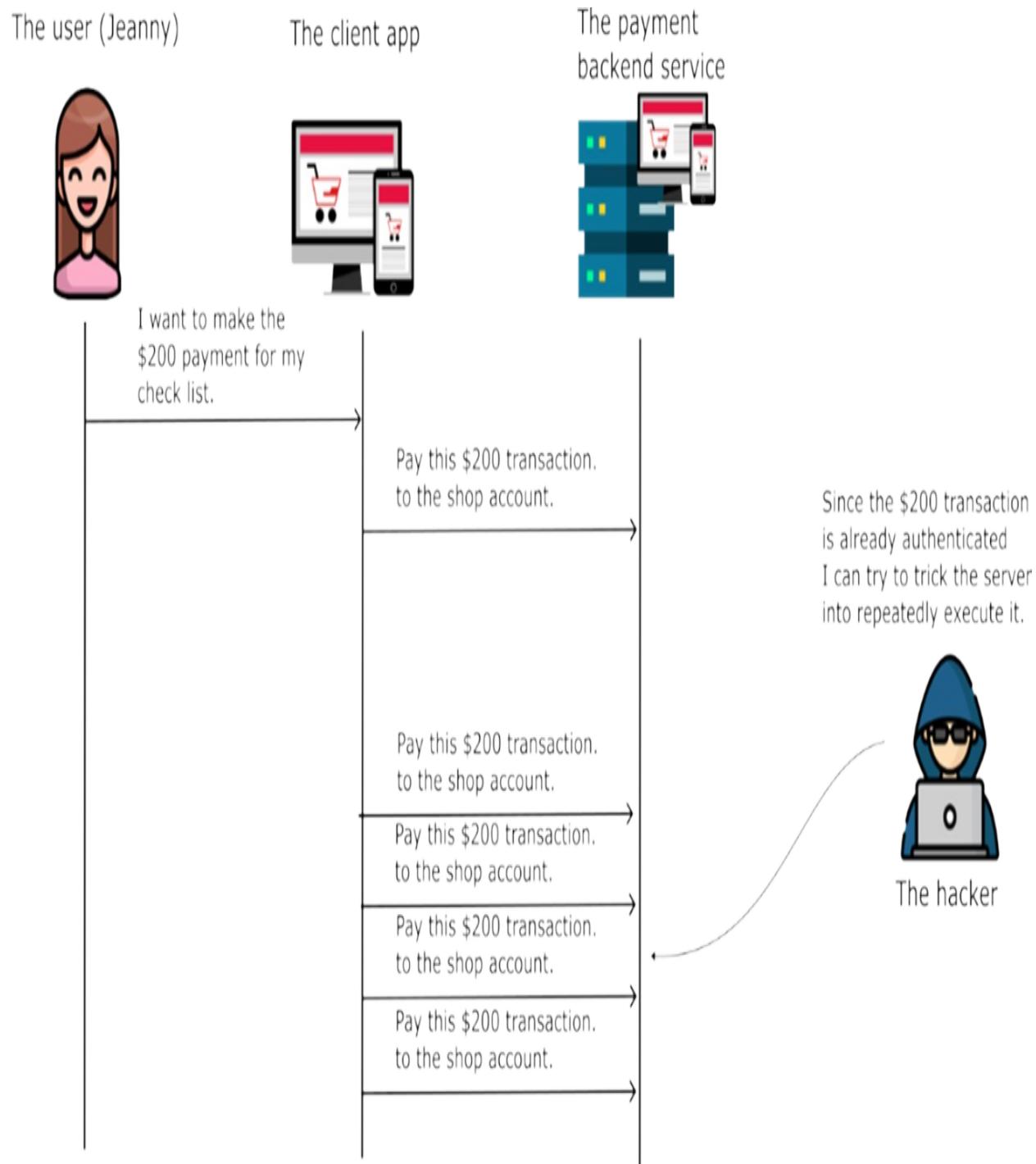
OpenID Connect (OIDC) enhances security by introducing built-in

mechanisms to defend against common web vulnerabilities, specifically replay attacks and Cross-Site Request Forgery (CSRF).

A replay attack happens when a hacker intercepts a valid piece of data—like a payment request—and resends it to trick the system into performing the same action again.

For example, imagine you make an online purchase, and your payment request is sent to the store's server to process the transaction. If an attacker intercepts that payment request, they could resend it multiple times, causing the store to charge your account again and again for the same order (figure 13.6). Without proper security checks in place, the system might treat these repeated requests as legitimate and continue processing unauthorized payments.

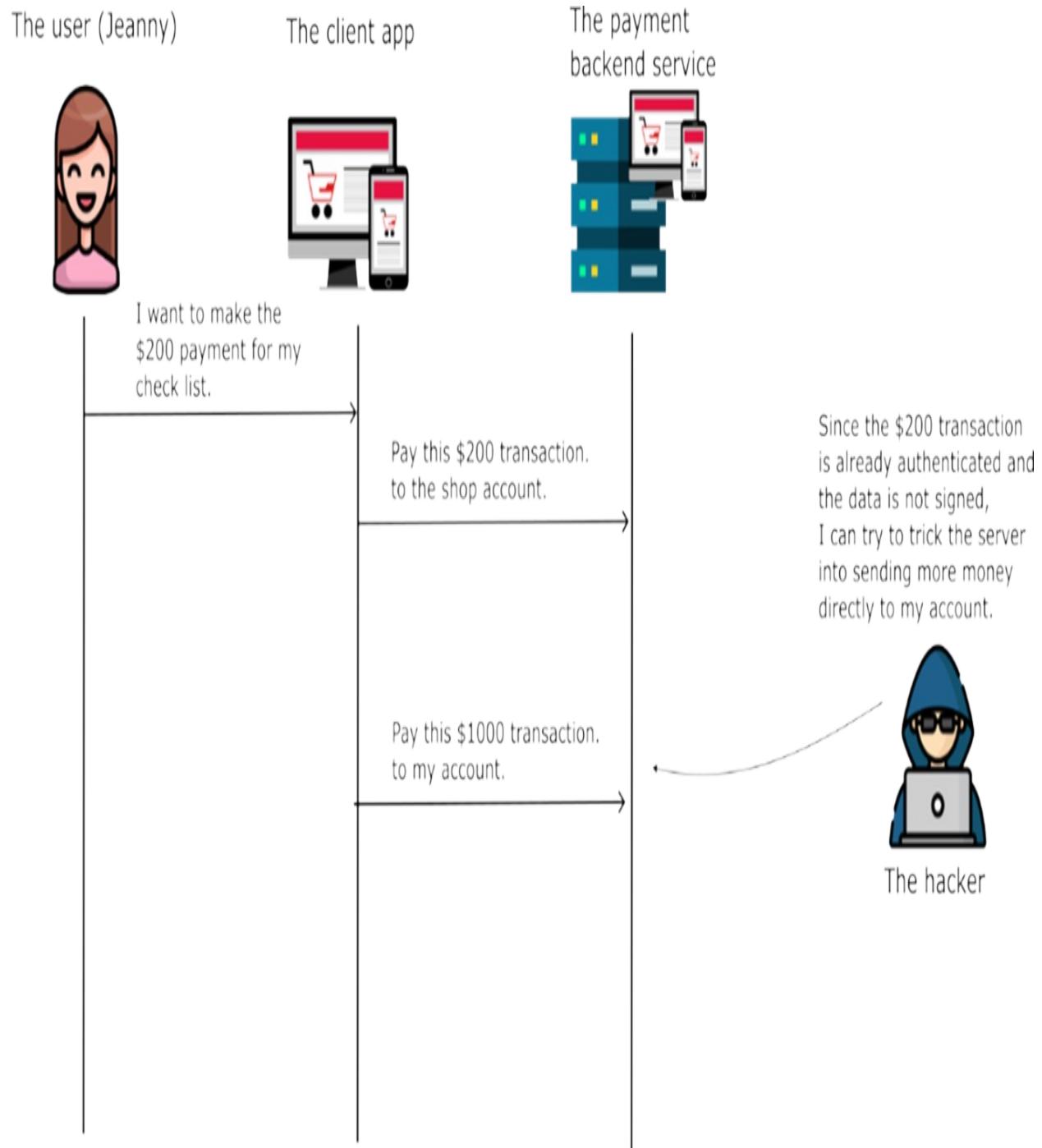
Figure 13.6 Illustration of a Replay Attack—An attacker intercepts and maliciously resends a previously valid request to trigger unauthorized actions or changes within the system.



If the system doesn't properly validate the integrity of the request, an attacker could potentially even modify it before replaying it. For example, in the case of a payment system, if the transaction data isn't securely signed or encrypted, a hacker could intercept the payment request and change the recipient's account details before resending it. This would trick the system

into sending the money to the attacker's account instead of the intended recipient (figure 13.7).

Figure 13.7 Example of a Replay Attack with Data Tampering—If cryptographic signing is not used, a hacker can intercept and modify a transaction. In this case, the attacker increases the transaction amount and redirects the payment to their own account instead of the legitimate shop's account.

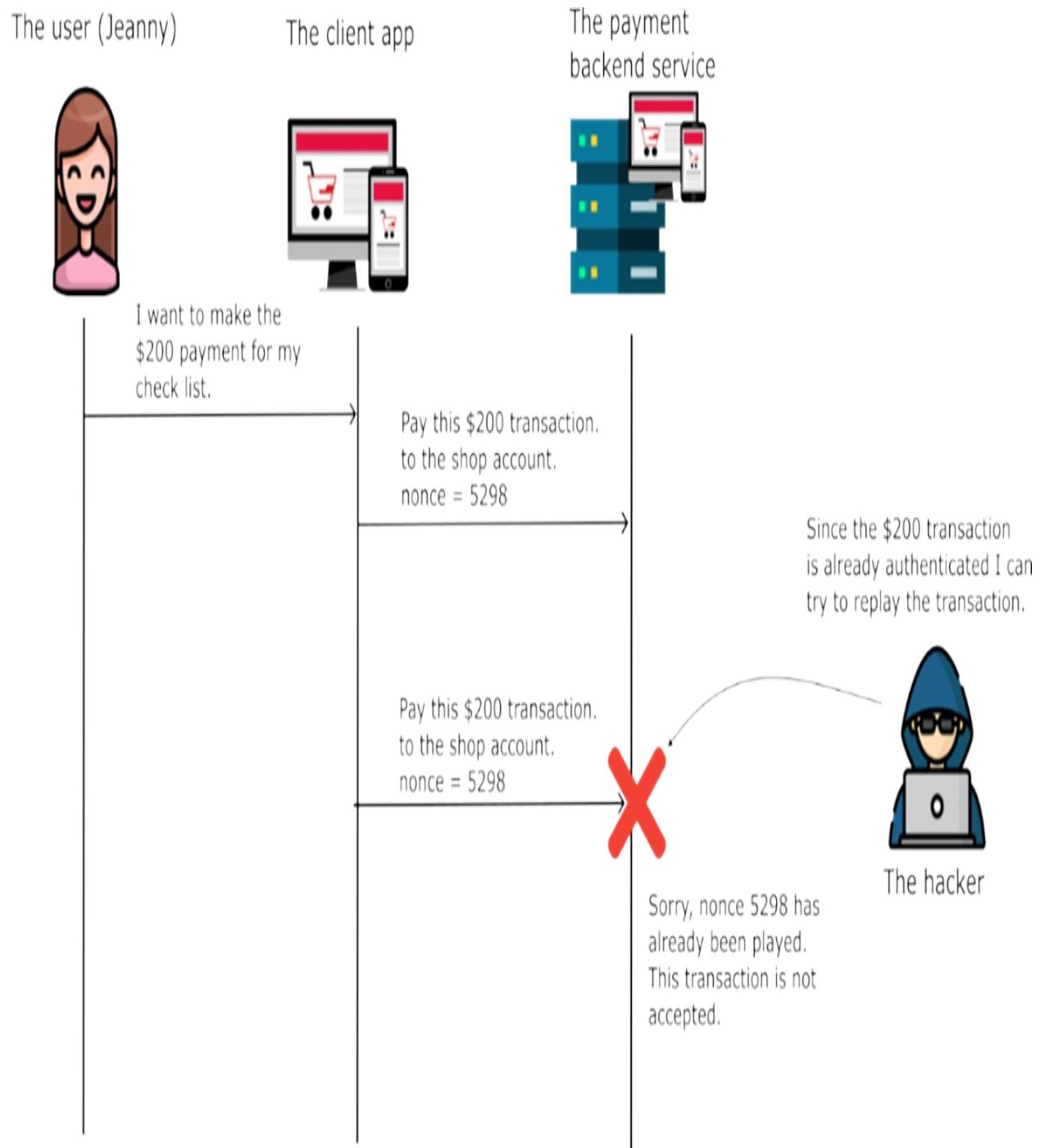


However, well-designed systems prevent this by:

- Digitally signing sensitive data to prevent tampering.
- Encrypting the request so it can't be read or altered in transit.
- Using nonces and timestamps to make each request unique and valid for a limited time.

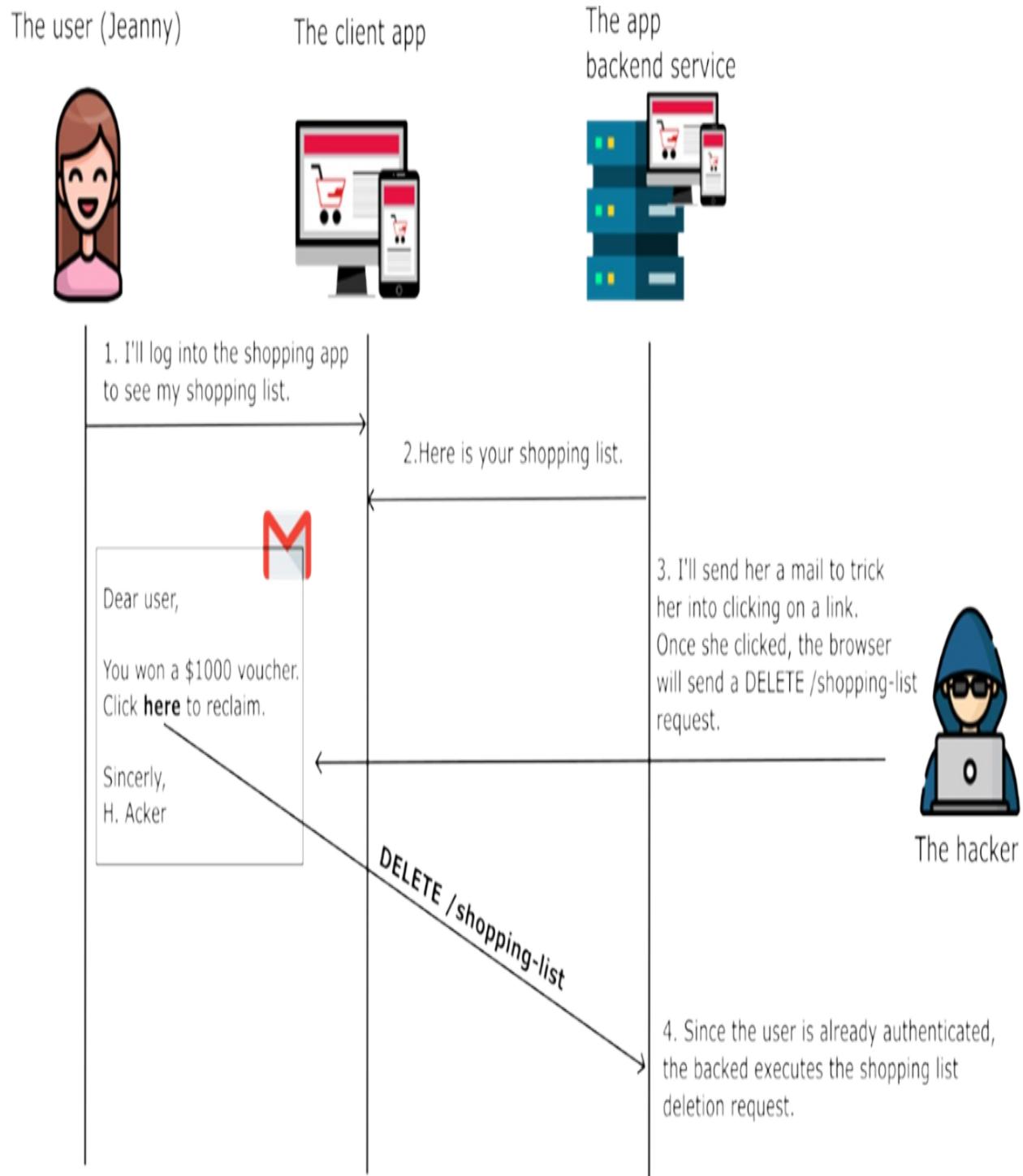
The *nonce* (Number Used Once) is a unique, random value generated by the client and included in the authentication request. With OpenID Connect, when the ID Token is returned, it must contain the same nonce value. This ensures the token is tied to the original request and prevents replay attacks by making each authentication response valid only once as shown in figure 13.8. If the nonce doesn't match, the response is rejected, effectively blocking the malicious reuse of authentication data.

Figure 13.8 Using a nonce with a properly signed transaction prevents hackers from replaying or modifying the request, ensuring the transaction is secure and unique.



The state parameter is another random value sent with the authentication request and returned unchanged by the authorization server. Its primary role is to prevent Cross-Site Request Forgery (CSRF) attacks by linking the request to the user's session. CSRF occurs when a malicious website tricks a user's browser into performing unintended actions on a trusted site where the user is already authenticated (figure 13.9).

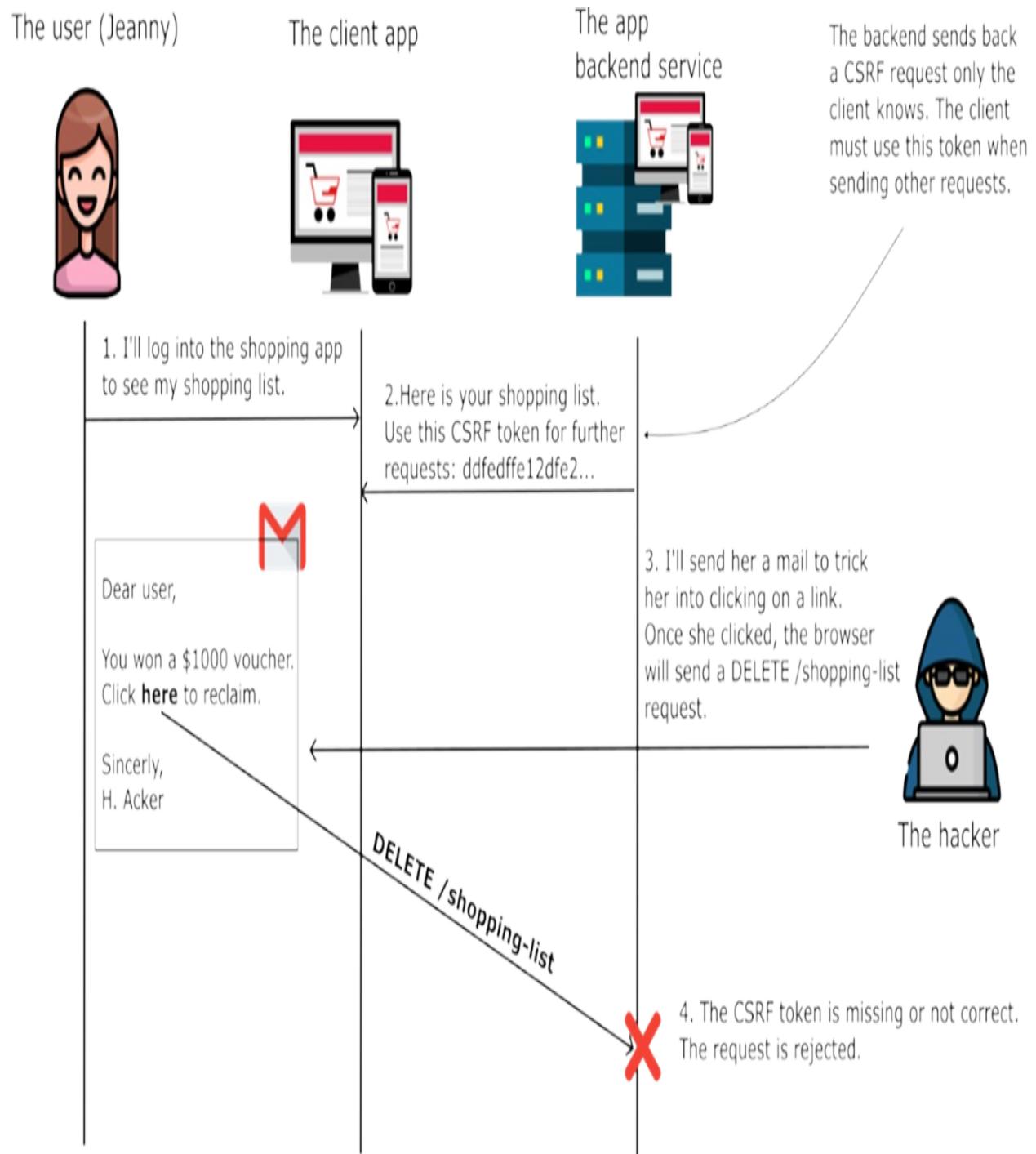
Figure 13.9 Cross-Site Request Forgery (CSRF)—An attacker tricks a logged-in user into unknowingly sending a malicious request to a trusted website. Because the user is already authenticated, the server treats the request as legitimate and executes unauthorized actions, potentially leading to unintended changes or data exposure.



For example, if a user is logged into their bank account, an attacker could

exploit CSRF to initiate unauthorized transactions without the user's consent. By verifying the state parameter, OIDC ensures that each authentication request is legitimate and originates from the correct client session, blocking such malicious attempts (figure 13.10).

Figure 13.10 Cross-Site Request Forgery (CSRF) Tokens—The server issues a unique, secret token to the client, which must be included in all subsequent requests. If the token is missing, invalid, or incorrect, the server rejects the request. Since the attacker does not know this token, they cannot craft a valid request, preventing them from tricking the user into executing unauthorized actions.



13.3.4 Session management and logout

OpenID Connect (OIDC) introduces standardized session management and logout mechanisms, making it easier for applications to handle user sessions securely and consistently across multiple services. This is a significant

improvement over OAuth 2.0, which leaves session handling entirely up to developers, often resulting in fragmented and insecure implementations.

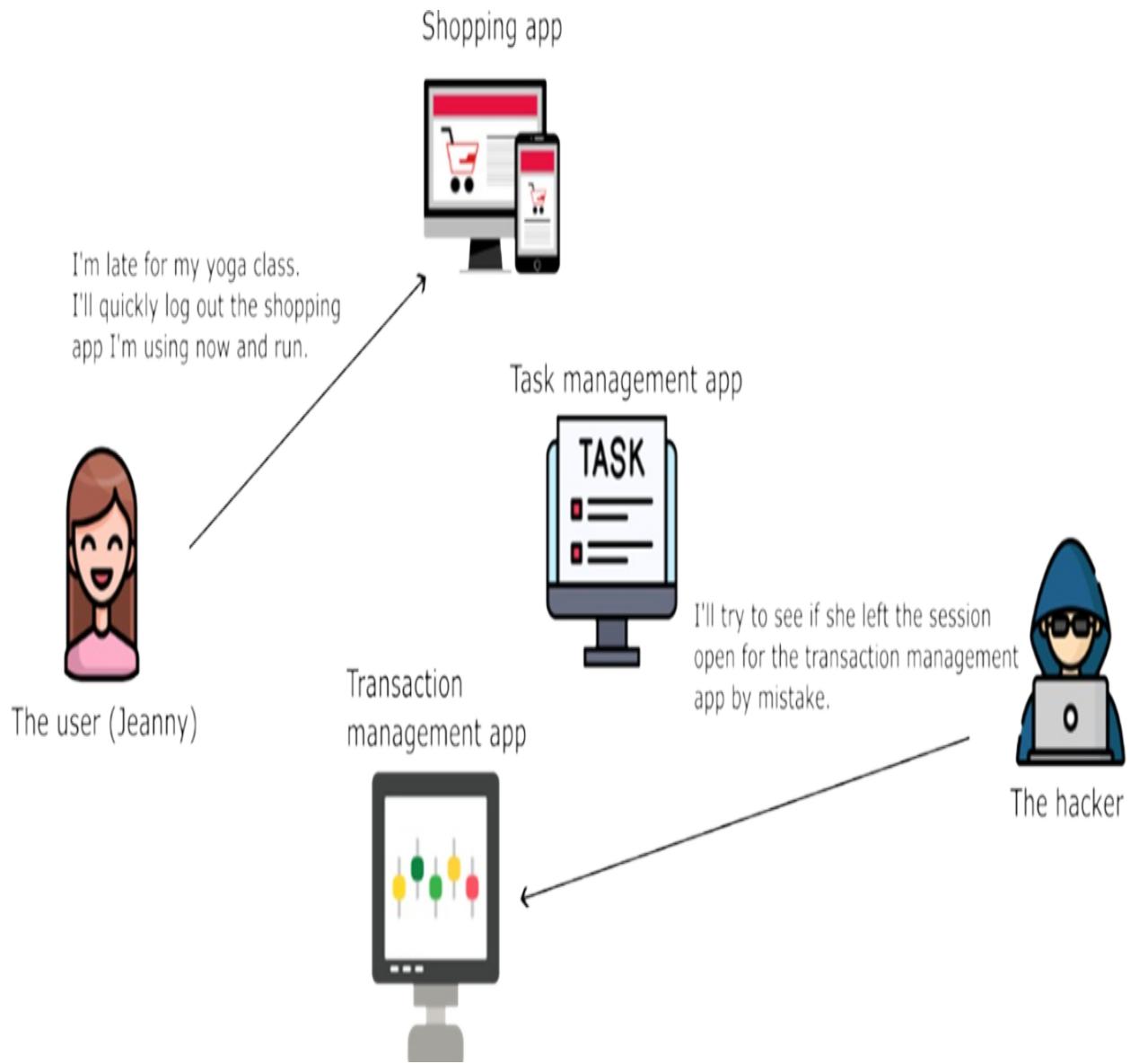
Session management ensures that user sessions are properly maintained and synchronized between the client application and the identity provider (IdP).

Imagine a user is logged into several apps through the same identity provider:

- Email Service
- Acme Inc.'s Online Storage Service
- Acme Inc. Project Management Tool

Without session management, logging out of the project management tool wouldn't affect the other two apps—the user would still be logged into their email and storage.

Figure 13.11 Without centralized session management, users may unintentionally leave multiple sessions open across different devices or applications, increasing the risk of unauthorized access and potential security breaches.



With OIDC's session management and logout, logging out from the project management tool triggers a logout request to the identity provider, which also logs the user out of the email and storage apps. Additionally, if the user's session expires on the identity provider, all connected applications recognize this and automatically sign the user out.

This consistent session handling improves security (by preventing forgotten active sessions) and enhances the user experience (with seamless login and logout across apps).

OIDC provides ways to check if a user is still logged in or if their session has

expired.

- *Session State Parameter* - OIDC introduces a session state parameter that tracks the user's login state. The client can silently check with the authorization server to see if the user is still authenticated without disrupting the user experience. If the session has expired, the client can prompt the user to log in again.
- *Silent Authentication (Prompt=none)* - OIDC allows the client to send a request with prompt=none, which checks the user's session in the background. If the session is still active, the user remains logged in; if not, the client can initiate a login flow.

OIDC also standardizes how applications handle user logout, addressing the issue of logging out across multiple connected applications.

- *RP-Initiated Logout* - The Relying Party (RP) (the client application) can trigger a logout by redirecting the user to the identity provider's logout endpoint. This ends the session at the IdP level and can redirect the user back to the application or another page after logging out.
- *Single Logout (SLO)* - OIDC supports Single Logout, where logging out from one application automatically logs out the user from all connected applications. This prevents situations where users think they've logged out, but other services remain active.

13.3.5 Exercises

8. What does OpenID Connect (OIDC) add to OAuth 2?
9. What's the ID Token?
10. Why is the ID Token always a JWT?
11. What's the UserInfo Endpoint for?

13.4 Using multitenancy

Multitenancy in the context of OpenID Connect (OIDC) is like running a massive apartment building where each tenant (client, organization, or application) has their own private space, keys, and decor—but the whole building shares the same security system, elevators, and utilities.

In technical terms, it means designing authentication and identity management systems that can securely and efficiently serve multiple tenants from a single Identity Provider (IdP). Each tenant operates independently, with their own users, roles, and settings, while still relying on the same underlying authentication infrastructure.

So, tenants get the freedom to paint their walls whatever color they want, but the building makes sure the locks work, the alarms are set, and no one sneaks into someone else's apartment!

As an enterprise, implementing multitenancy with OpenID Connect (OIDC) is highly beneficial in several scenarios, especially if your organization delivers services or manages systems for multiple clients, departments, or business units. Here's where and why you'd use it:

- *Software-as-a-Service (SaaS) Platforms* - If your enterprise offers a SaaS product to multiple companies or customers, multitenancy allows you to serve all clients through a single platform while keeping their data, users, and access controls separate.
- *Large Enterprises with Multiple Business Units* - If your company manages IT services for multiple clients, multitenancy lets you provide secure, tailored authentication for each client from a unified system.
- *Customer and Partner Portals* - Enterprises often have different portals for customers, vendors, and partners. Multitenancy allows separate authentication and access control for each group.

Let's revisit our Acme Inc. scenario to make things clearer.

As Acme Inc. grew into a leading enterprise, it rapidly expanded into an online retail giant, selling everything from high-tech gadgets to stylish home decor. However, as the business scaled, Acme encountered a significant challenge: managing suppliers, handling logistics, and tracking inventory became increasingly complex—not just for them, but for many other online retailers facing the same struggles.

Recognizing this industry-wide pain point, Acme Inc. decided to turn its operational expertise into a business opportunity. They launched Acme Commerce Cloud, a powerful Software-as-a-Service (SaaS) platform

designed to help e-commerce businesses streamline inventory management, supplier coordination, and order processing.

Now, Acme Inc. offers this platform to a variety of online retailers, each with unique security requirements and authentication systems. Let's explore how Acme Inc. leverages OIDC multitenancy to securely and efficiently serve these diverse clients.

Among Acme Inc's clients we have now the following three tenants:

- *Trendy Threads* - a fashion e-commerce startup that uses Google Workspace for all its internal operations, including employee accounts.
- *MegaMart Online* - a large online retailer that sells everything from groceries to electronics. Given their size and sensitivity of customer data, they use Microsoft Azure Active Directory (Azure AD) for secure authentication and enforce strict security policies like Multi-Factor Authentication (MFA).
- *CozyHome Decor* - a small online store selling handcrafted home decor items. They don't have a dedicated IT team and prefer a simple, built-in authentication system.

As you can see, all three tenants have distinct business models and security requirements—a common scenario in the real world where each organization follows its own business strategies and IT policies. Trendy Threads wants its employees to log into Acme Commerce Cloud using their existing Google accounts to keep things simple and seamless. MegaMart Online prioritizes security and compliance, insisting on managing authentication through Azure AD to meet industry regulations and enforce strict access controls. CozyHome Decor seeks an easy-to-use login system managed entirely by Acme Commerce Cloud, eliminating the need for technical expertise.

By implementing OIDC multitenancy, Acme Commerce Cloud caters to these diverse needs:

One of the points of interest is flexible authentication, which involves dynamic routing of login requests to different identity providers (Google Workspace, Azure AD, or Acme's native system). Acme Commerce Cloud routes login requests from Trendy Threads to Google Workspace

(<https://idp.acmecommercecloud.com/trendythreads/authorize>), while MegaMart employees are redirected to Azure AD for login (<https://idp.acmecommercecloud.com/megamart/authorize>).

Each tenant will also prioritize a specific customer experience, requiring personalized and branded login pages tailored to their identity. Employees from Trendy Threads see a login page featuring their company's logo and brand colors, creating a seamless and familiar experience. Similarly, MegaMart's login page is fully customized with their branding, and critical security policies like Multi-Factor Authentication (MFA) are automatically enforced. Meanwhile, CozyHome employees access a clean, Acme-branded login page with simple email/password login options and optional social logins for added flexibility.

The three tenants will likely implement different authorization strategies, defining unique roles, privileges, and access controls based on their business needs. These roles and permissions may be named and structured differently across tenants. Acme Commerce Cloud can handle this by embedding tenant-specific claims in the issued tokens. For example, the following code snippet shows how the user info response might look for a user from MegaMart.

```
{  
  "sub": "user567",  
  "tenant_id": "megamart", #A  
  "role": "inventory_manager",  
  "permissions": ["view_reports", "manage_suppliers"]  
}
```

Acme Inc. must also carefully manage session handling to ensure security and a smooth user experience. For example, if Emma from Trendy Threads logs out, it should not impact John from MegaMart. Each tenant's session must remain fully isolated to prevent any cross-tenant interference. Fortunately, the OIDC protocol offers robust solutions for managing these challenges when implementing multitenancy.

To conclude, let's go through some key concepts you should understand and remember about multitenancy with OIDC:

- *Tenant isolation* - Each tenant should have isolated identity data,

configurations, and security policies. This prevents one tenant from accessing or interfering with another tenant's data.

- *Dynamic client registration* - OIDC supports dynamic client registration, allowing tenants to register their own client applications with the identity provider. This makes scaling across tenants easier without manual intervention.
- *Tenant-Specific Endpoints* - Authentication requests and token exchanges can be routed through tenant-specific endpoints. For example, for “Tenant 1” the app would use <https://idp.example.com/tenant1/authorize> while for “Tenant 2” the app would use <https://idp.example.com/tenant2/authorize> as the authorization endpoint. This keeps each tenant's authentication flow separate.
- *Custom Branding and User Experience* - The identity provider can serve customized login pages, branding, and user flows for each tenant. OIDC enables this by allowing tenant-specific configuration for login UI, consent screens, and user journeys.
- *Tenant-Aware Claims and Scopes* - OIDC allows the inclusion of tenant-specific claims in the ID Token or responses from the UserInfo Endpoint. This helps client applications distinguish between users from different tenants.
- *Role-Based Access Control (RBAC) per Tenant* - Each tenant can define its own roles and permissions. The identity provider can include these roles in the ID Token or Access Token, enabling tenant-specific authorization.
- *Isolated Session Management* - OIDC can manage user sessions separately for each tenant, ensuring that a logout from one tenant's app doesn't affect sessions in another tenant's environment.

13.4.1 Exercises

12. What's multitenancy in identity systems?
13. How does OIDC help with multitenancy?
14. Can different tenants use different login methods?
15. How do you avoid users mixing between tenants?

13.5 Answers to exercises

1. What is PKCE and why is it important?

PKCE (Proof Key for Code Exchange) protects public clients, like mobile or web apps, from attackers who might steal an authorization code and use it to get access.

2. How does PKCE work?

Before authentication, the client app generates a random secret (the code verifier) and sends a hashed version of it (the code challenge) with the request. Later, it must show the original secret to get the token.

3. Why is the code challenge secure?

Because it's made using a cryptographic hash function like SHA-256, and it's nearly impossible to reverse it back to the original secret.

4. When should you use PKCE?

Always with public clients that can't safely store secrets, such as single-page apps, mobile apps, or IoT devices.

5. What are refresh tokens for?

They let apps get new access tokens without asking the user to log in again—keeping sessions smooth and secure.

6. Why not make access tokens last longer?

Long-lived tokens are risky if stolen. Short-lived tokens reduce that risk, and refresh tokens help maintain usability.

7. How do you protect refresh tokens?

Store them securely, rotate them after use, set expiration times, and revoke them on suspicious activity.

8. What does OpenID Connect (OIDC) add to OAuth 2?

OIDC adds authentication. It answers “Who is the user?” while OAuth 2 just says “What can they access?”

9. What's the ID Token?

It's a special token that contains user identity information (like email, name, etc.) and proves who just logged in.

10. Why is the ID Token always a JWT?

Because it must contain readable claims. JWTs are perfect for securely packaging identity data in a standard way.

11. What's the UserInfo Endpoint for?

It gives you extra user profile data, in case you need more than what's in the ID Token.

12. What's multitenancy in identity systems?
It means one identity system can serve multiple clients or organizations, keeping each tenant's users and data separate.
13. How does OIDC help with multitenancy?
It supports tenant-specific endpoints, claims, branding, and roles, so each client gets a custom experience.
14. Can different tenants use different login methods?
Yes! One can use Google, another Azure AD, and another your native system. OIDC handles the routing.
15. How do you avoid users mixing between tenants?
By using tenant-specific login URLs and issuing tokens with tenant-specific claims (like tenant_id).

13.6 Summary

- Proof Key for Code Exchange (PKCE) strengthens the Authorization Code Flow by preventing authorization code interception, especially for public clients like mobile and web apps.
- Refresh Tokens allow applications to maintain user sessions without repeatedly prompting for login, balancing security (by limiting access token lifespan) and user experience (by reducing interruptions).
- OpenID Connect's Identity Layer introduces the ID Token, securely providing user identity information alongside the access token, enabling applications to authenticate users safely and reliably.
- The UserInfo Endpoint allows applications to retrieve additional user profile information in a standardized and secure manner, complementing the ID Token.
- OIDC enhances security against Replay Attacks and Cross-Site Request Forgery (CSRF) by using security parameters like nonce and state, ensuring requests are legitimate and unique.
- Session Management and Logout are standardized in OIDC, allowing consistent user session handling and Single Logout (SLO) across multiple connected applications, improving both security and user experience.
- Multitenancy with OIDC enables Acme Commerce Cloud to securely serve multiple tenants with distinct authentication flows, custom branding, isolated sessions, and tenant-specific roles and permissions—

- all from a single identity provider.
- Best Practices include always using PKCE with public clients, securely storing refresh tokens, enabling refresh token rotation, and enforcing secure communication with TLS/SSL.

14 Passwordless login: Using Magic links and OTPs

This chapter covers

- Using magic links
- Using one-time passwords (OTPs) for authentication
- Protecting your apps from passwordless authentication vulnerabilities

You're trying to log into your favorite app, but you can't remember your password. Was it your dog's name plus your birth year? No, wait—that was your banking account. Maybe it's P@ssw0rd123? Oh no, you changed it last month! After five failed attempts and a CAPTCHA test that makes you question your ability to recognize traffic lights, you finally give up and hit "Forgot Password."

And just like that, passwords have once again defeated you.

But relax! Passwordless authentication is here to rescue you from the endless cycle of password resets and security questions like, "What was the name of your childhood best friend's second cousin's goldfish?"

In this chapter, we'll explore the magic (literally) of *Magic Links*—think of them as the Portkeys of authentication, instantly transporting you into your account. Then, we'll dive into the thrilling world of One-Time Passwords (OTPs), the digital equivalent of casting "Alohomora" to unlock your login.

But the passwordless adventure doesn't end here. Finally, in chapter 15, we'll unveil the high-tech sorcery that is WebAuthn—where your face, fingerprint, or a tiny security key acts like a personal wand, proving that you, and only you, are the true master of your account.

14.1 The real magic of magic links authentication

Alice was on a mission: She had found the perfect pair of shoes on Acme.com, and nothing would stop her from buying them. She eagerly navigated to the login page, but something was... off.

Instead of the usual "Username" and "Password" fields, there was just one lonely box asking for her email.

"Wait a minute," Alice muttered, double-checking her screen. "Where's the password field? Did Acme forget it? Have I been hacked? Is this some sort of trap?"

She hesitated, feeling the ghost of a thousand forgotten passwords haunting her. But then, curiosity won. She typed in her email and hit "Log In."

Within seconds, a new email arrived:

"Click this link to log in to your Acme account!"

No password. No security questions. Just one click.

Alice stared at the email, then back at her screen. Was this sorcery? A trick? Had Acme secretly hired wizards to make her life easier?

She clicked the link.

Boom. She was in!

And just like that, Alice realized - passwords were a thing of the past.

Let's now dive into the implementation of magic link authentication. Below are the key steps that make it work securely and efficiently (figure 14.1):

1. *User Requests a Login Link* – The user enters their email on the login page, triggering the authentication process.
2. *App Generates a Secure, Time-Limited Magic Link* – A unique, cryptographically signed token is created and added to a link, ensuring that the link is valid only for a short period and cannot be easily forged or reused.
3. *App Sends the Magic Link via Email* – The system delivers the login link

to the user's email, ensuring it is tamper-proof, properly formatted, and expires after a set duration (usually a few minutes).

4. *User Clicks the Link* – The users expresses their intention to enter the application.
5. *The Redirect and Authentication* - Once the user clicks the link, the app validates the token, checks for expiration. It establishes a secure session, granting the user access without requiring a password.

Figure 14.1 Authenticating through magic links. Upon the user's request, the app generates a unique link and sends it to the user by email. The user clicks on the link to authenticate.

The user

Acme. Inc.

Email App



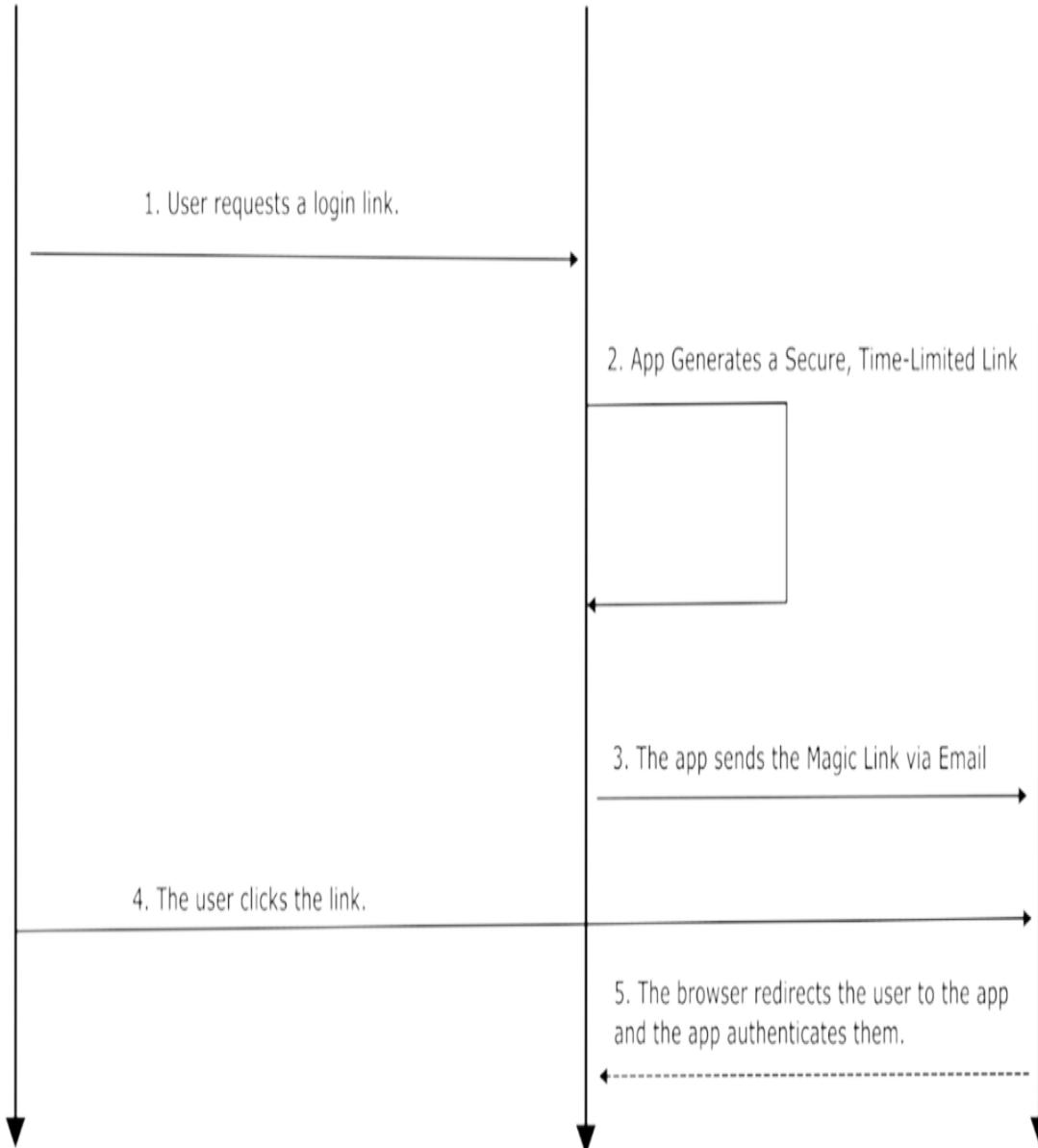
1. User requests a login link.

2. App Generates a Secure, Time-Limited Link

3. The app sends the Magic Link via Email

4. The user clicks the link.

5. The browser redirects the user to the app
and the app authenticates them.



The magic link authentication process begins when the user enters their email on the login page. The system then checks whether the email exists in the database. If it's a valid, registered email, the process moves forward to generate a secure login link. However, if the system doesn't find the email, the system doesn't reveal this fact directly. Instead, it returns a generic message like, "If an account exists, we've sent a login link," to prevent attackers from using the login page to guess which emails are registered. This small but crucial detail helps protect user privacy and minimizes the risk of email enumeration attacks.

An *email enumeration attack* occurs when an attacker systematically tests whether specific email addresses (or usernames) exist on a system by observing how the application responds to login, signup, or password reset attempts. If the system provides different responses based on whether an account exists, the attacker can confirm valid accounts and use this information for further attacks, such as brute-force logins, phishing, or credential stuffing.

Note

One of the most critical security principles in credential-based authentication is never disclosing whether an account exists. Messages like "The password is incorrect" might seem harmless, but they can unintentionally confirm to an attacker that they've found a valid username or email. This information leakage gives hackers one more piece of the puzzle, making it easier to target specific accounts, attempt credential stuffing, or launch password reset attacks. Instead, always use generic error messages like "Invalid credentials" or "If an account exists, we've sent a login link" to avoid unintentionally assisting attackers in their efforts.

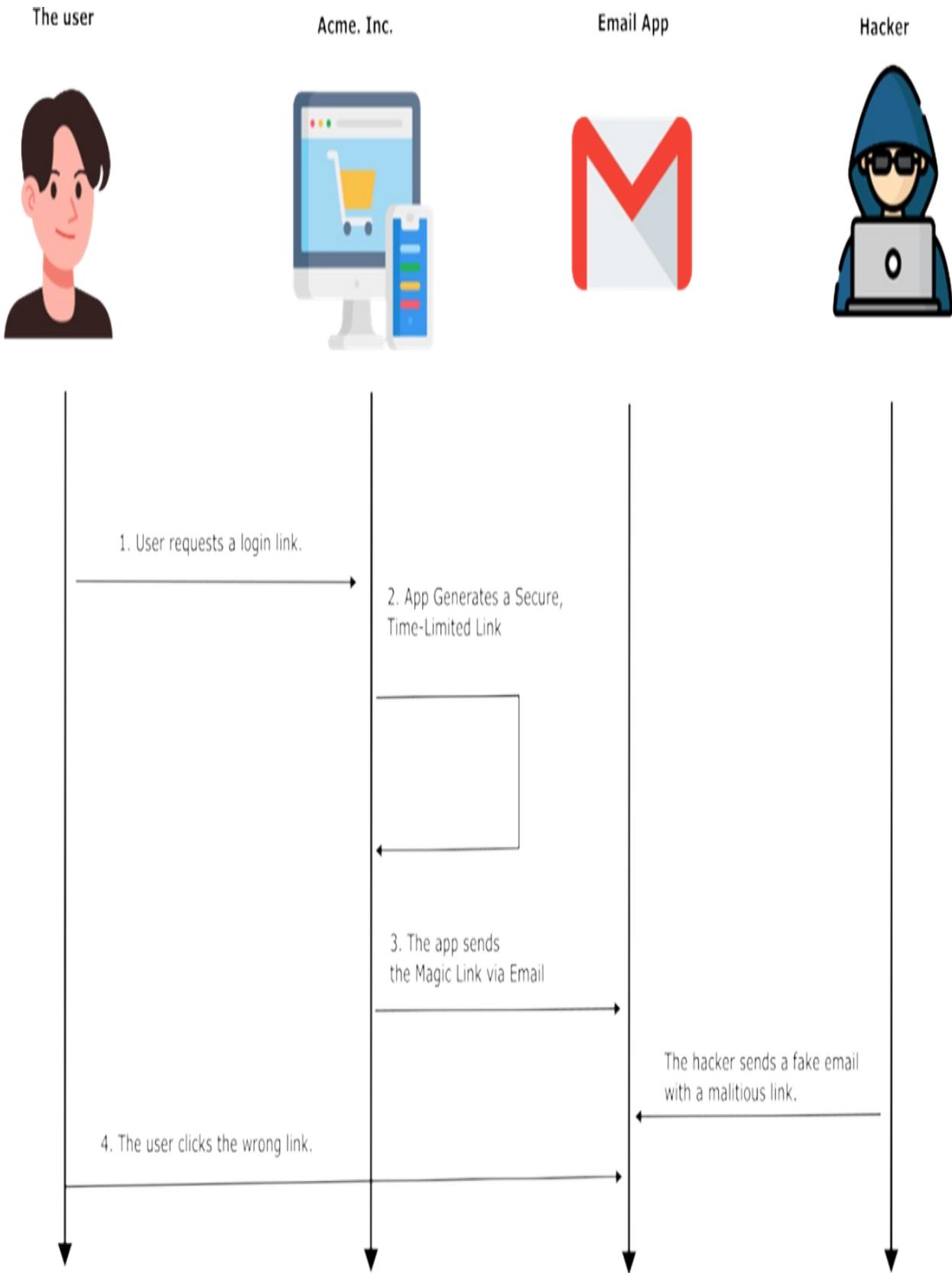
You might wonder, why is this so important? What can attackers really do with just a username or an email?

Well, quite a lot—and none of it is good. Once hackers confirm that an email exists, they can target high-value accounts like corporate emails or admin users, making them the perfect victims for more advanced attacks. If you've ever reused a password (and let's be honest, most people have), they can try

credential stuffing—plugging your email and old passwords into different websites, hoping to get lucky. If they find a match - they're in. Another thing they can do is targeted phishing. If they know you have a Spotify account, they can send a more realistic fake email, tricking you into clicking an unwanted link.

Now, let's take it a step further—what if attackers combine email enumeration with magic links? Once they know an email is valid, they can send convincing fake emails pretending to be from your app, saying something like, "Here's your secure login link for Acme. Inc.! Click to sign in." If the email looks real enough—maybe even spoofing the sender address —there's a good chance the user will click the fraudulent link without thinking twice. Instead of logging in, they unknowingly hand over their authentication token or get redirected to a fake login page that steals their credentials (figure 14.2).

Figure 14.2 A malicious individual might send a fake but realistically looking email between steps 3 and 4 of the process. An inattentive user could be fooled into clicking the wrong link.



Since magic links remove the need for passwords, users might let their guard down, assuming “it’s secure because I don’t have to type anything.” But in reality, if an attacker tricks you into using their fake magic link, they can intercept your login request and gain access to your account without ever needing your password. This is why email security, link verification, and proper domain checks are just as crucial as authentication. Also, *multi-factor authentication (MFA)* is usually an excellent plus for an app’s authentication process. We’ll discuss MFA later, in section 14.2. After all, even the most secure system can’t protect users if they’re tricked into handing over the keys themselves.

Security is a lot like an alarm system—it doesn’t matter how expensive or high-tech it is if it’s poorly installed or not used correctly (figure 14.3). You could have the best locks, motion sensors, and surveillance cameras, but if you leave a window open, an intruder can just walk right in. The same goes for authentication: even if magic links or other security measures are designed to be safe, they can be exploited if users aren’t careful—like clicking a fake login link from a phishing email.

As we discussed in previous chapters, security isn’t just about having the right tools; it’s about implementing them properly and ensuring users don’t unknowingly bypass them. Even the most advanced system is only as strong as its weakest link—and if that link is human error or a poorly configured setting, attackers will find a way in.

Figure 14.3 Security is only as strong as its weakest link. You can have the best defenses, but if you leave the key under the rug... well, good luck!



Once the user requests a magic link, the system must generate a secure, one-time-use token to verify their identity. This token acts like a temporary digital key, allowing users to log in without a password. To ensure security, the

token must be cryptographically signed - typically using JWT (JSON Web Token) or HMAC-based encryption—so that it can't be easily forged or manipulated.

You might want to take a break and revisit chapter 6, where we explored JSON Object Signing and Encryption (JOSE) and the fundamentals of JWT (JSON Web Tokens). Understanding how JWTs work - how they're signed, verified, and securely transmitted - will help you grasp the mechanics behind magic links and why they are both powerful and potentially risky if not implemented correctly.

The token should include essential claims such as the user's email or ID, an expiration timestamp (e.g., 10 minutes), and a nonce (a random value that prevents replay attacks, ensuring the token can't be reused maliciously—remember we discussed nonce in chapter 13 as well).

```
{  
  "email": "user@example.com",  
  "exp": 1715179200,  
  "nonce": "random-string-here"  
}
```

Without storing the token, the system cannot invalidate it once it's used. This means that if an attacker somehow intercepts or obtains the magic link before the user clicks it, they might be able to reuse it.

With DB storage, the system can mark the token as "used" immediately after the first successful login. If someone tries to use the same token again, the system rejects it. If a user reports suspicious activity, the system can immediately revoke their magic link by deleting or invalidating the token in the database.

Attackers can brute-force magic link generation by repeatedly requesting links for different emails. If the system doesn't store tokens, it won't be able to detect excessive login attempts from the same IP or email or limit how often a user can request a magic link. The system can throttle requests and prevent abuse by keeping a record of issued tokens.

Another best practice is auditing and security monitoring for generated magic links. For example, the system can trigger an alert if multiple tokens are generated for the same email within a short period. At the same time, if a magic link is used from an unexpected location or device, the system can require additional verification.

Once the magic link token is generated, the system constructs a secure login URL containing the token.

<https://acmeinc.com/auth/magic?token=eyJhbGciOiJIUzI1NiIsInR5cCI6>

This link is then emailed to the user (step 3), but several security precautions must be taken to prevent misuse. First, the link should expire quickly - typically within 5 to 15 minutes - to minimize the risk of unauthorized access if intercepted. Second, the email should include clear branding and familiar design elements to ensure users recognize it as a legitimate communication from your app, reducing the risk of phishing attacks.

Lastly, the link should never be included in plaintext within the email body; it should only exist within the clickable link to prevent accidental exposure or leaks.

If the link (or attached token) appears as plain text in the email, the user might accidentally copy and share it—for example, by pasting it in a public chat. Some email clients preview text content before the user even opens the email. If the token appears in the email body, it might be cached or exposed in notifications, increasing the risk of unauthorized access. If someone else gets access to the token, they can use it to log in as the user without needing their email or password.

Moreover, many email services and security tools automatically scan, log, or archive email contents. If the token is in plain text, it might be stored or indexed in logs where an attacker—or even a malicious insider—could retrieve it later.

When the user clicks the magic link (step 4), they are redirected to your application, where the system extracts the authentication token from the URL

and begins the validation process. First, it checks the token's signature (if using JWT or HMAC) to ensure it hasn't been tampered with. Then, it verifies whether the token has expired, rejecting any that are no longer valid. If the system stores the tokens, it also ensures that the token hasn't already been used to prevent replay attacks.

If all checks pass, the user is logged in - either by creating a session or issuing a new authentication token (such as a fresh JWT or session ID). For added security, the system can invalidate the magic link after first use, ensuring an attacker cannot reuse it. Finally, the user is redirected to their dashboard or the page they originally intended to visit, completing the passwordless authentication flow seamlessly and securely.

Table 14.1 summarizes the best practices you have to remember about magic links.

Table 14.1 Best practices for magic links implementations.

Best practice	Why?
Use short-lived tokens (5–15 min expiration)	Minimizes the window for attackers to use a stolen or leaked token.
Store tokens in a database or cache	Prevents token reuse, enables manual revocation, and allows tracking of authentication attempts.
Invalidate used tokens after	Stops replay attacks by ensuring a token can only be used once.
Use cryptographically signed tokens	Ensures tokens cannot be forged or manipulated, adding security against tampering.
Never send tokens in plaintext within the email	Prevents accidental exposure if the email is forwarded or intercepted.
Ensure magic links include clear	Reduces phishing risks by making emails easily recognizable as legitimate.
Limit how often a user can request a magic	Protects against automated attacks that try to flood the system with requests.

Check token validity (signature, expiration, uniqueness)	Prevents expired, tampered, or duplicate tokens from being used for authentication.
Warn users about suspicious activity	Detects and notifies users of potential unauthorized login attempts.
Allow fallback authentication	Provides a way for users to access their account if the magic link email is delayed or blocked.

And there you have it - passwordless authentication through magic links! A system so smooth, it feels like you just waved a wand and logged in. No more forgetting passwords, no more resetting them every time you take a vacation, and definitely no more shouting “I swear I used the right one!” at your screen.

But like all magic tricks, there’s a catch—if you don’t secure the system properly, someone else might pull the rabbit out of the hat. Expired tokens, phishing attacks, and email leaks can turn your sleek login experience into a hacker’s best friend. So while magic links make life easier, they’re only as good as the security practices behind them.

Now, if only we could magic away those "Verify you're not a robot" CAPTCHAs, we'd truly be in the future – or does these actually put us in the future? Who knows?

14.1.1 Exercises

1. How does magic link authentication work?
2. Why should the app not reveal if an email exists?
3. What are the security risks of magic links?
4. Why store magic link tokens in a database?
5. Why shouldn't the link appear in plaintext in the email?

14.2 Authentication through One-Time Passwords (OTPs)

One-Time Passwords (OTPs)—those little six-digit codes that always seem to arrive at the worst possible moment. You’re trying to log in, feeling confident, when suddenly your screen says:

"We've sent a code to your phone. Please enter it below."

And just like that, the waiting game begins. You check your phone. Nothing. You stare at it, willing the SMS to appear. Still nothing. Maybe your signal dropped? Maybe your carrier is taking a lunch break? You try resending the code, and just as you do, the first one arrives. Great. But wait—which code is the right one now? You take a guess, and of course—"Invalid OTP. Request a new one."

Welcome to the chaotic world of OTPs. They can be frustrating, sure, but they're also one of the most widely used security measures—helping prevent unauthorized access even if someone has your password. In this section, we'll explore how OTPs work, when they're useful, and the security risks they pose if not implemented correctly. So keep your phone nearby (hopefully with signal this time), and let's dive in!

Despite their occasional inconvenience and delayed arrivals, One-Time Passwords (OTPs) provide several key security advantages in authentication. Here's why they remain one of the most widely used methods for protecting user accounts:

- *They Are Time-Sensitive and Single-Use* - Unlike traditional passwords, OTPs expire quickly (usually within 30 seconds to a few minutes) and can only be used once. Even if an attacker intercepts an OTP, it will likely be useless by the time they try to use it.
- *They Work Without a Persistent Secret* - Unlike passwords, which users store, reuse, or write down on sticky notes, OTPs don't require users to remember anything. This means there's nothing static for attackers to steal.
- *They Can Be Delivered in Multiple Ways* - Users can receive OTPs via SMS, email, or authentication apps (like Google Authenticator, Authy, or Microsoft Authenticator). This makes them versatile and easy to deploy across different platforms.
- *They Don't Require Users to Create or Remember Another Password* -

Let's face it: people are terrible at remembering passwords. OTPs eliminate this problem by acting as a temporary login key, reducing the need for users to create and store complex passwords.

Let's analyze the steps for OTP authentication (figure 14.4):

1. *User requests an OTP for authentication* - The user enters their email or phone number to request a one-time password. The system checks if the provided contact is registered to prevent enumeration attacks.
2. *The app generates a secure OTP* – The app creates a random, time-limited OTP (usually 6–8 digits).
3. *The app sends the OTP to the user* - The OTP is delivered via SMS, email, or an authenticator app.
4. *The user enters the OTP and authenticates* - The system verifies the OTP for correctness, expiration, and single-use. If valid, the user is authenticated.

Figure 14.4 The OTP authentication process. The app generates a random OTP and sends it to the user. The user uses the received OTP to authenticate.

The user

Acme. Inc.

User's Phone



1. User requests an OTP for authentication.

2. App Generates a random OTP

3. The app sends the OTP to the user's phone

4. The inserts the OTP code for authentication.

When a user wants to log in with an OTP, they start by entering their email or phone number on the login page. The system then checks whether the provided information is registered to prevent unauthorized access attempts. If the email or phone number is valid, the system generates a one-time password and sends it to the user, allowing them to proceed with authentication.

Once the system verifies the user's email or phone number, it generates a random numeric code, usually 6 to 8 digits long. To ensure security, this OTP must be cryptographically random, making it impossible to predict or guess. The system then assigns an expiration time—typically between 30 seconds to a few minutes—to prevent attackers from using an old OTP to gain access. For validation, the OTP is temporarily stored, preferably in a hashed format, ensuring it remains secure and cannot be easily retrieved.

Unlike magic links (we discussed in section 14.1), which contain a self-contained token (often a signed JWT) embedded in the link itself, OTPs require user interaction—the user must receive, read, and manually enter the code. While magic links simplify authentication with a single click, OTPs introduce an extra step but provide flexibility since they can be delivered via multiple channels, such as SMS, email, or authenticator apps.

After receiving the OTP via email or SMS, the user enters it into the login page. The system then validates the OTP, checking if it is correct and still within its expiration window. If the OTP is incorrect, the login attempt is rejected, and if it has expired, the user is prompted to request a new one. Since OTPs are designed for one-time use, they are removed from the database after validation to prevent reuse. If the OTP is valid and matches the one generated, the system authenticates the user, granting access and establishing a session by either creating a session ID or issuing a new authentication token (such as a JWT).

A key best practice for securing OTP authentication is to implement rate limiting to prevent brute-force attacks. Users should be restricted to a reasonable number of OTP requests—for example, no more than five attempts per minute—to stop attackers from flooding the system with guesses. Additionally, the system should monitor login attempts for

suspicious behavior, such as multiple failed OTP entries or repeated requests from different locations, and flag unusual patterns for review. These measures help prevent abuse while ensuring a secure and controlled authentication process.

One Time Passwords and Multi-Factor Authentication

You might have encountered the OTP concept in other contexts as well, such as when discussing about multi-factor authentication (MFA). In high-risk scenarios (e.g., logging into sensitive accounts), consider requiring One-Time Passwords (OTPs) as part of Multi-Factor Authentication (MFA), alongside a primary password or another authentication factor, such as biometrics or security keys.

MFA is a security mechanism that requires users to provide two or more independent factors to verify their identity. Instead of relying solely on a password, MFA ensures that even if one factor is compromised, an attacker still cannot gain access. These authentication factors typically fall into three categories:

- Something You Know – A password, PIN, or security question.
- Something You Have – A smartphone, security token, or OTP.
- Something You Are – Biometrics like a fingerprint, face scan, or voice recognition.

By combining multiple factors, MFA significantly reduces the risk of unauthorized access, making it much harder for attackers to compromise accounts, even if they obtain a user's password.

While OTPs can be used as a standalone authentication method, they are often implemented as one factor in an MFA setup. In a typical MFA process:

1. The user enters their username and password (something they know).
2. The system verifies the credentials and then prompts for an OTP (something they have).
3. The user retrieves the OTP from SMS, email, or an authenticator app and enters it into the login page.
4. If both the password and OTP are correct, authentication is granted.

By requiring an OTP in addition to a password, the system ensures that even if an attacker steals the password (e.g., through phishing or credential stuffing), they still cannot log in without access to the user's OTP device.

Magic links and OTPs might look very similar, in the end both aim to make authentication more secure and convenient by eliminating the need for passwords. But while they share a common goal, they work differently and have unique strengths and weaknesses. Table 14.2 compares the two:

Table 14.2 A comparison between OTPs and Magic Links.

Feature	OTPs	Magic Links
How It Works	User receives a temporary code (via SMS, email, or an app) and enters it manually.	User receives a unique link via email, clicks it, and is logged in automatically.
User Effort	Requires manually entering a code.	Just click a link—no typing needed.
Security Risks	SMS OTPs can be intercepted (SIM swapping, SS7 attacks). Phishing attacks can trick users into entering OTPs on fake sites. Brute force attacks possible without rate limiting.	If an attacker gains access to a user's email, they can log in easily. Phishing emails can mimic real magic links. If links aren't expired after first use, they can be reused.
Speed & User Experience	Can be slow if the OTP is delayed or expires too quickly.	Faster since it's a single click (but depends on email delivery speed).
Dependency	Relies on SMS, email, or an authenticator app.	Relies entirely on email security and availability.
Multi-Factor Authentication (MFA)?	Can be used as a second factor in MFA.	Usually a single-factor authentication method.
Phishing Resistance	Users might be tricked	Users might be tricked

	into entering OTPs on fake sites.	into clicking fake login links.
Best For	Securing accounts with MFA. Apps that support TOTP (Google Authenticator, Authy). Users with unreliable email access.	Passwordless login experiences. Users who struggle with passwords. Websites where authentication needs to be quick and seamless.

Table 14.3 summarizes the best practices we discussed for using OTPs.

Table 14.3 Best practices for using One-Time Passwords (OTPs).

Best practice	Why?
Use short-lived OTPs (30 sec to 5 min)	Reduces the window of opportunity for attackers to reuse or intercept an OTP.
Generate OTPs using a cryptographically secure method	Ensures OTPs cannot be guessed or predicted, making brute-force attacks ineffective.
Limit OTP requests to prevent brute-force attacks	Prevents automated attacks from flooding the system with OTP requests.
Use OTPs only once and invalidate them after use.	Prevents replay attacks where an attacker tries to reuse a stolen OTP.
Encourage the use of app-based OTPs over SMS/Email.	App-based OTPs (like Google Authenticator) are more secure than SMS, which is vulnerable to SIM swapping and SS7 attacks.
Monitor login attempts for suspicious behavior	Helps detect potential attacks, such as multiple failed OTP entries from different locations.
Provide fallback authentication methods	Allows users to log in securely if they don't receive an OTP (e.g., backup codes or alternative authentication methods).

Implement rate-limiting on OTP entry attempts.	Stops attackers from attempting multiple OTP guesses by locking out excessive attempts.
--	---

To conclude our discussion on One-Time Passwords (OTPs), it's important to recognize that while they enhance security compared to static passwords, they still have vulnerabilities, such as phishing and man-in-the-middle attacks. For an even more secure, phishing-resistant authentication method, we explore WebAuthn in chapter 15, where we discuss how it enables passwordless authentication using public-key cryptography and biometric or hardware-based authenticators.

14.2.1 Exercises

6. What makes OTPs more secure than passwords?
7. What are OTP security risks?
8. How do OTPs fit into multi-factor authentication (MFA)?
9. Build a Simple Magic Link Authentication Flow (Mocked).

Implement a minimal version of the magic link flow using your preferred language or framework. You don't need to send real emails, just simulate the flow locally.

Requirements:

- A simple webpage or command-line interface where the user enters an email address.
- A function that:
 - Generates a magic link with a JWT token (include email, expiration, and a random nonce).
 - Stores the token in an in-memory store.
- A second function simulating the click of the link:
 - Verifies the token signature and expiration.
 - Invalidates the token after the first use.

Bonus:

- Add protection against token reuse.

- Log every login attempt with metadata (IP address if available, timestamp).
- Add a throttling mechanism to limit how often a user can request a link.

14.3 Answers to exercises

1. How does magic link authentication work?

The user enters their email, gets a login link by email, and clicks it to log in, no password needed.

2. Why should the app not reveal if an email exists?

If it does, attackers can find out who's registered and target them.
Always use generic messages like "If your email exists..."

3. What are the security risks of magic links?

Phishing attacks can trick users with fake emails. If links aren't short-lived or invalidated after use, attackers might reuse them.

4. Why store magic link tokens in a database?

To detect replay, limit how often a user can request one, and let the system invalidate them after they're used.

5. Why shouldn't the link appear in plaintext in the email?

It could be leaked, cached, or seen by others. Always hide it behind a proper clickable button.

6. What makes OTPs more secure than passwords?

They expire quickly, are used only once, and don't need to be remembered—there's nothing to reuse or guess later.

7. What are OTP security risks?

If sent by SMS, they can be intercepted (e.g., via SIM swapping). If rate-limiting is missing, attackers can guess them.

8. How do OTPs fit into multi-factor authentication (MFA)?

They're often the "something you have" part, combined with a password or biometric to boost account protection.

9. Build a Simple Magic Link Authentication Flow (Mocked)

<<<URL to solution>>

14.4 Summary

- Passwordless authentication improves security and convenience by

eliminating weak or reused passwords. This chapter explored Magic Links, OTPs, Biometric Authentication, and Hardware Security Keys as alternatives. Each method has its advantages and risks.

- Magic links allow users to log in via a one-time link sent to their email. However, they are vulnerable to phishing and email compromise. Using short-lived, cryptographically signed tokens and preventing email enumeration helps mitigate risks.
- OTPs provide temporary numeric codes via SMS, email, or authenticator apps. They are susceptible to SIM-swapping, phishing, and brute-force attacks. Using app-based OTPs (Google Authenticator, Authy) and limiting OTP requests increases security.
- Passwordless authentication reduces phishing, credential leaks, and login friction, but no single method fits all use cases. Choosing the right approach depends on security needs, usability, and risk factors.
- Proper implementation is key—even the most secure authentication method can be exploited if configured poorly. Following best practices, encryption standards, and secure handling of authentication data ensures a stronger, safer login experience.

15 Passwordless login: WebAuthn and hardware authentication

This chapter covers

- Using biometric authentication
- Using hardware keys for authentication
- Protecting your apps from passwordless authentication vulnerabilities

Imagine this: You’re trying to log in, but instead of typing a password and playing the “forgot password” game, you just look at your phone, tap a key, or scan your finger - You’re in! No passwords to forget, no SMS codes to wait for, and no hacker guessing your childhood pet’s name (RIP Fluffy).

But how do these futuristic authentication methods actually work? Are they really as secure as they sound? And what happens if you shave your beard or lose your security key? In this section, we’ll explore the magic behind biometric authentication and hardware security keys, their strengths, their potential weaknesses, and why they might just be the future of secure logins.

Biometric authentication and *hardware security keys* are passwordless authentication methods that verify a user’s identity in a highly secure and convenient way. Instead of relying on something you know (like a password), they rely on something you are (biometrics) or something you have (a physical security key).

Biometric authentication and hardware security keys are passwordless authentication methods that verify a user’s identity in a highly secure and convenient way. Instead of relying on something you know (like a password), they rely on something you are (biometrics) or something you have (a physical security key).

One of the most widely adopted standards enabling passwordless authentication is *WebAuthn* (Web Authentication), developed by the FIDO

Alliance and standardized by the W3C. WebAuthn allows websites and applications to authenticate users using public-key cryptography instead of traditional credentials. It supports authentication through biometrics (such as fingerprint or facial recognition) and security keys (such as FIDO2-compliant hardware tokens or built-in authenticators on modern devices).

As we'll discuss in this chapter, WebAuthn enhances security by eliminating the risks associated with passwords, such as phishing, credential stuffing, and brute-force attacks. Since authentication is tied to a specific device and involves cryptographic key pairs, attackers cannot reuse stolen credentials elsewhere. Additionally, WebAuthn is supported by major web browsers and operating systems, making it a scalable and user-friendly solution for passwordless authentication. We'll talk all the essentials you need to know in this chapter, but if you're curious about all the details, you find the official documentation related to WebAuthn at the following link:

<https://www.w3.org/TR/webauthn/>

15.1 Biometric authentication

A core part of WebAuthn is biometric authentication, which leverages unique biological traits to verify identity. Since biometric data is stored securely on the user's device and never transmitted, it significantly reduces the risk of credential theft. Additionally, WebAuthn ensures that each authentication is bound to a specific device, preventing attackers from using stolen credentials elsewhere. This method uses unique biological traits to confirm a user's identity. Common biometric authentication types include:

- Fingerprint Scanners – Used on smartphones, laptops, and security systems.
- Facial Recognition – Unlocks devices and accounts with a quick face scan.
- Iris or Retina Scans – Used in high-security environments for precise identification.
- Voice Recognition – Identifies users based on speech patterns.

While many biometric methods exist, some are more widely used in consumer and enterprise applications:

- Fingerprint scanning
- Facial recognition
- Iris and retina scanning
- Voice recognition
- Palm vein recognition

15.1.1 Fingerprint Scanning (Most Common)

Fingerprint scanning is one of the most widely used biometric authentication methods, found in smartphones, laptops, and smart cards. It uses fingerprint sensors to capture and compare unique fingerprint patterns, ensuring quick and reliable authentication. Its biggest advantages are speed, widespread adoption, and strong security. However, it does have some weaknesses - high-quality fingerprint replicas can sometimes trick the system, and sensor wear over time may lead to false rejections or reduced accuracy. Despite these challenges, fingerprint scanning remains a trusted and efficient method for biometric authentication.

15.1.2 Facial Recognition (Rapidly Growing)

Facial recognition has become a popular biometric authentication method, widely used in smartphones (Face ID), laptops (Windows Hello), and security systems. It uses infrared depth sensors or AI-based 2D/3D mapping to scan and verify a user's face. Its key advantages include speed and hands-free authentication, making it both convenient and seamless. However, its security depends on the technology used - systems without 3D depth sensors can be vulnerable to photo or video spoofing, making them less reliable in high-security environments. Despite this, facial recognition continues to grow as a fast and user-friendly authentication method.

15.1.3 Iris & Retina Scanning (Highly Secure)

Iris scanning is a highly secure biometric authentication method, commonly used in high-security environments and select smartphones, such as Samsung's Iris Scanner. It works by analyzing the unique patterns in a person's iris, making it extremely difficult to spoof—even more so than fingerprint or facial recognition. Additionally, it remains effective even if a

user's fingerprint is damaged or unreadable. However, its adoption is limited due to its need for specialized hardware and the fact that it is generally slower than other biometric methods. Despite these drawbacks, iris scanning remains one of the most secure biometric authentication technologies available.

15.1.4 Voice Recognition (Used in Call Centers & Virtual Assistants)

Voice recognition is a convenient biometric authentication method, commonly used in banking call centers and voice assistants like Siri and Google Assistant. It analyzes a user's unique voice characteristics to verify their identity. The biggest advantages of voice recognition are that it's hands-free and highly accessible, making it useful for users with disabilities or those who prefer a simple authentication experience. However, it also has significant security risks—it can be spoofed using voice recordings or AI-generated voices, making it less secure than fingerprint or facial recognition in high-risk scenarios.

15.1.5 Palm Vein Recognition (High Security, Less Common)

Palm vein recognition is a highly secure biometric authentication method, primarily used in ATMs, secure facilities, and healthcare authentication systems. It works by scanning the unique vein patterns inside the palm using infrared sensors, making it nearly impossible to replicate or spoof. This method offers exceptional security, as vein patterns are internal and not easily accessible like fingerprints or facial features. However, its adoption is limited due to the need for specialized hardware, and it is not as widely available as other biometric methods. Despite this, palm vein recognition remains an excellent choice for high-security applications.

Biometric authentication is fast and user-friendly but comes with concerns like privacy risks and false rejections (e.g., your phone doesn't recognize you before coffee).

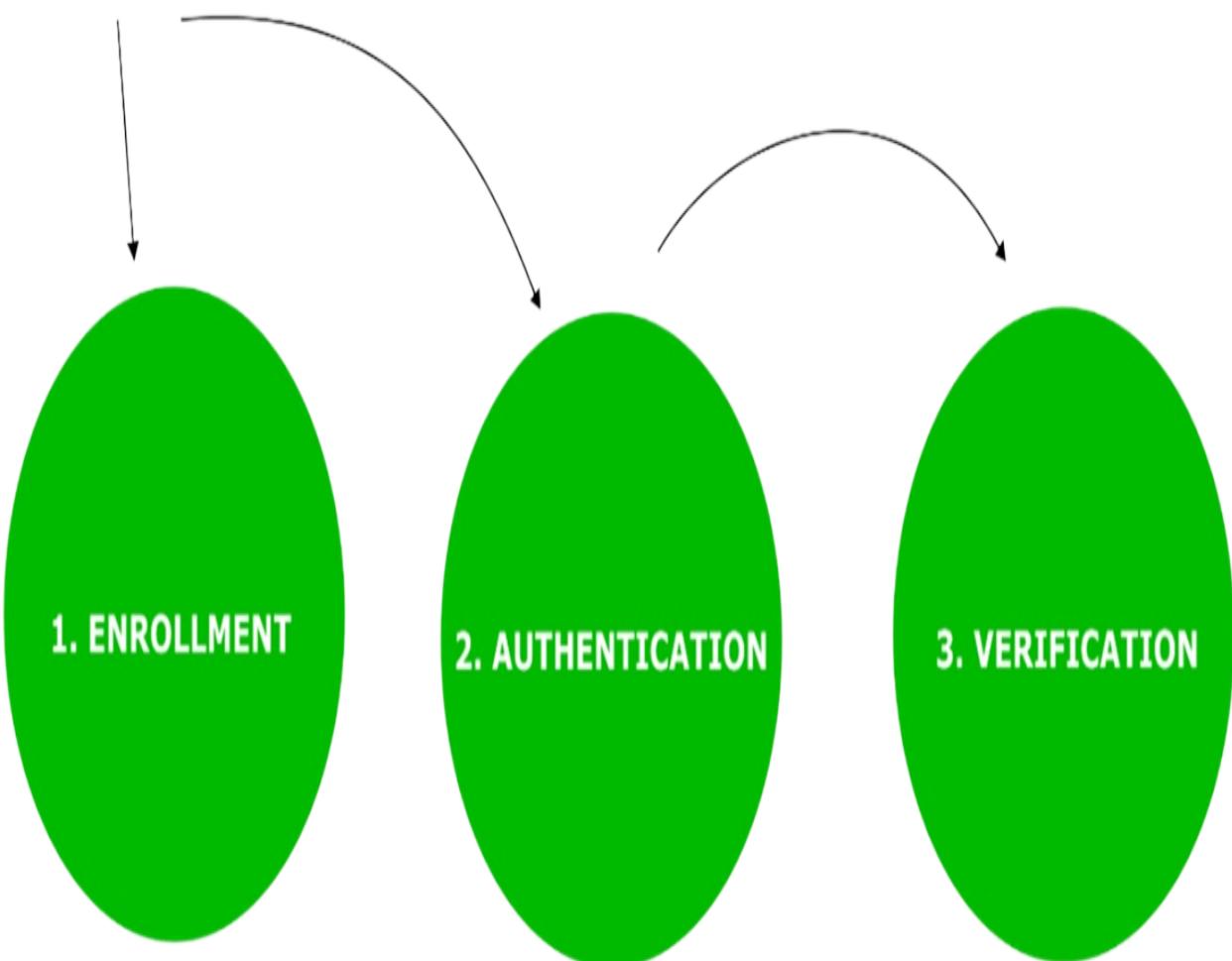
Integrating biometric authentication into an app involves securely verifying a user's identity using a device's built-in biometric sensors (like fingerprint readers or face scanners). The essential steps to enable biometric

authentication imply:

1. *User enrollment* - the user registers their fingerprint, face, or other biometric data on their device.
2. *Authentication request* - The user attempts to log in to the app. Instead of a password, the app prompts for biometric authentication (e.g., fingerprint scan, face recognition).
3. *Verification process* - The device checks the user's biometric input against the previously stored biometric data. If it matches, the system generates a cryptographic authentication token that confirms the user's identity. With a valid token, the app grants access - either by logging the user in or allowing sensitive actions (e.g., approving a payment).

Implementing biometric authentication follows a simple process involving enrollment, authentication, and verification (figure 15.1). While it provides a fast and secure way to log in, it is not without vulnerabilities.

Figure 15.1 A three-step process. Initially, a user needs to register their data. Following registration, the user can try to authenticate and get verified by the system.



Problems may appear even from the very first step – the user enrollment. Let's first understand how it works: When a user sets up biometric authentication, they register their fingerprint, face, iris, or voice on their device. The system then captures the unique features and securely stores them in a *Trusted Execution Environment (TEE)*, such as Apple's Secure

Enclave, Android's TEE, or a hardware-backed TPM (Trusted Platform Module). Instead of saving raw biometric data, the system converts it into a mathematical template, creating a hashed representation of the biometric pattern.

Think of biometric authentication as making a unique key for a lock. Instead of storing the exact shape of the key, the system creates a blueprint that describes its key pattern in a way that only the lock can understand.

Or imagine you press your fingerprint into soft wax to create a seal. Instead of keeping the wax itself, you take a picture of it and store only the measurements of the ridges and curves. Later, when you try to unlock a door, you press your real fingerprint into another wax mold. The system doesn't compare the exact fingerprint but instead checks whether the new measurements match the stored ones. If they do, you're granted access!

This is exactly how biometric authentication works! The system doesn't store your actual fingerprint, face, or iris. Instead, it converts it into a mathematical representation (a template). When you try to log in, the system checks for a close match, not an identical one.

Now let's review some vulnerabilities that can happen considering this approach.

Picture David, a seasoned hacker. He had pulled off password leaks before, but this time, he was after something even more valuable—biometric templates. He knew that unlike passwords, biometric data can't be changed—if he could steal it, victims would be vulnerable forever.

One day, David found a company storing biometric templates in an unsecured database. Jackpot! Instead of encrypted tokens, he found raw biometric templates, mathematical representations of users' fingerprints and facial features. With this data, David didn't need to trick users into handing over passwords—he could generate fake biometric scans that matched the stolen templates, giving him access to their accounts as if he were them.

The worst part? Users couldn't reset their faces or fingerprints as they could with passwords. Once stolen, their biometric data was permanently

compromised. David knew that if companies didn't store templates securely—inside hardware-protected areas attackers like him would have a gold mine of unchangeable identities to exploit.

Even if template stealing would not work, David still has other ways to exploit user registration for biometric data. Assume an application that, instead of properly verifying the user before letting them register a fingerprint or face scan, simply trusted whatever was submitted.

Using deepfake technology (which is no longer something difficult to get your hands on), David created an ultra-realistic AI-generated face that mimicked real users' biometric patterns. Then, he enrolled the fake face into an account, essentially registering himself as an authorized user. Now, whenever he wanted access, he didn't need to hack anything—he could just use his fake biometric profile, and the system would treat him as a legitimate user.

The beauty of this trick? The actual user would never know. Even if they later tried to log in, the system wouldn't alert them that an extra face or fingerprint had been registered. David had created a permanent backdoor, bypassing passwords, OTPs, and even multi-factor authentication.

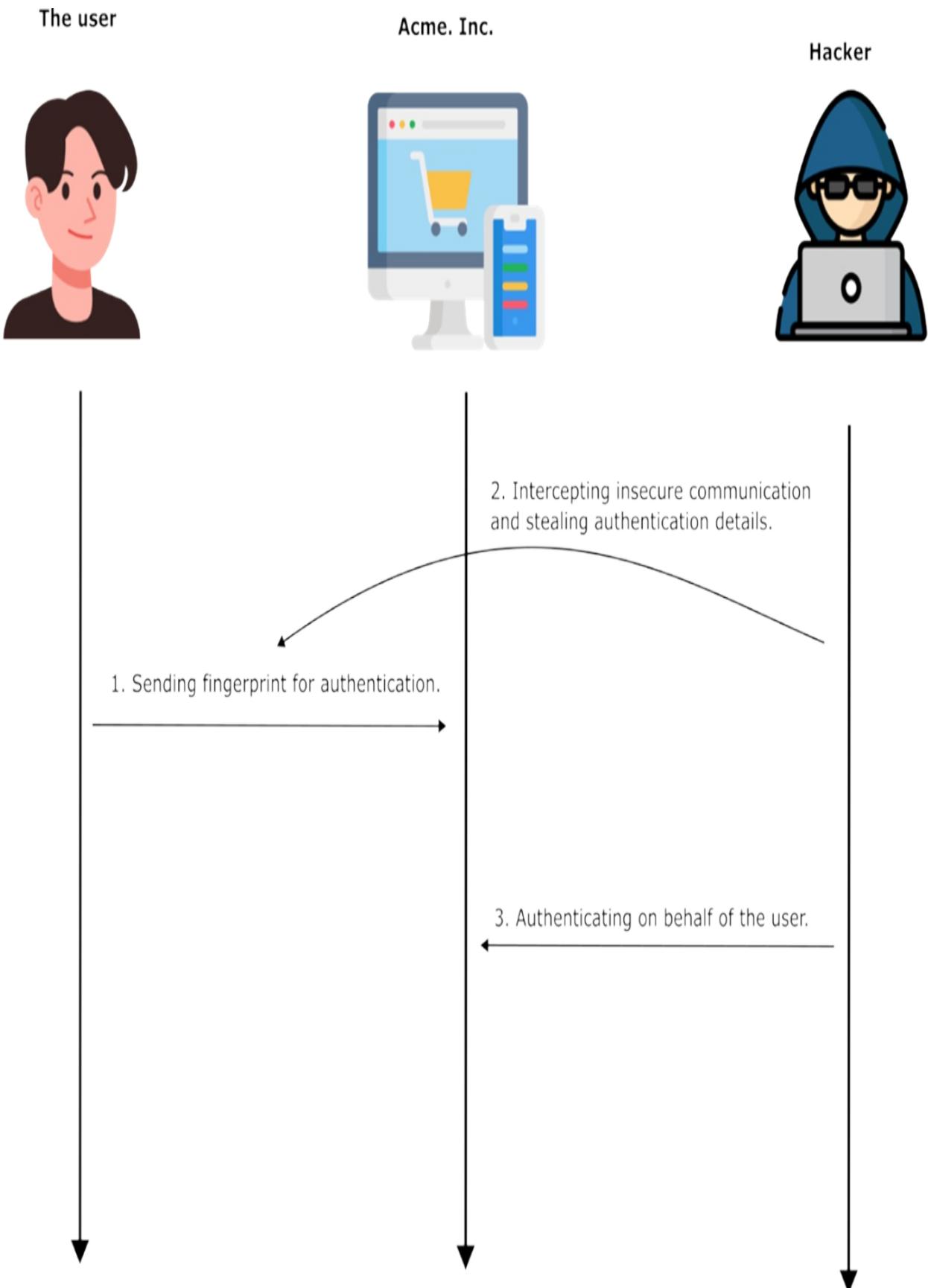
If companies didn't enforce strict identity verification before allowing biometric enrollment, attackers like David could register fake or stolen biometric data, granting them undetectable, long-term access to high-value accounts.

But the user registration isn't the only one that could have vulnerabilities if the developers didn't consider a strong security implementation. The authentication process itself might have its own flows.

One method is a replay attack, where a hacker captures and replays biometric authentication data to trick the system into granting access (figure 15.2). Another serious threat is a man-in-the-middle (MITM) Attack, where biometric data is intercepted if transmitted unencrypted over a network, allowing an attacker to spoof authentication remotely.

Figure 15.2 Improperly secured transmission and storage of biometric authentication data can

expose it to hackers, who may then exploit it in replay attacks.



But not all attacks require hacking - Phishing via Social Engineering can be just as dangerous. A well-crafted message like “Please scan your fingerprint to confirm this transaction” can trick users into authenticating a malicious request without realizing it. These vulnerabilities highlight why biometric authentication must be properly implemented with encryption, anti-replay protections, and strong user awareness.

Biometric systems are designed to recognize unique physical traits, but attackers have found ways to fool the technology. Spoofing attacks involve using high-quality fake fingerprints, 3D face masks, or AI-generated voices to trick biometric sensors into granting access. A similar method, presentation attacks, relies on showing the system a photo, video, or fingerprint mold - a common weakness in poorly secured facial recognition systems that lack liveness detection. But even if the biometric scan is secure, attackers can sometimes bypass authentication entirely by stealing security tokens. If malware or session hijacking allows an attacker to capture the cryptographic token generated after successful biometric authentication, they can impersonate the user without ever needing their fingerprint or face scan.

If you’ve ever watched a spy movie, you’ve probably seen a scene where the bad guys cut off someone’s finger to bypass a fingerprint scanner, or maybe they pluck out an eyeball to fool an iris scanner. Classic Hollywood hacking, right? Well, thankfully, real-life biometric security isn’t that easy to break—modern systems use liveness detection to check for things like blood flow, muscle movement, and blinking to ensure that the biometric input is actually attached to a living person (sorry, bad guys).

But while finger-snatching villains may not be a real-world threat, biometric authentication still has its weaknesses. Spoofing attacks, stolen biometric templates, and deepfake-generated faces can all create vulnerabilities if the system isn’t designed with proper security measures. Unlike passwords, you can’t just change your face or fingerprints if they get compromised—so companies implementing biometrics must get it right the first time.

At the end of the day, biometric authentication is incredibly powerful, offering both security and convenience. But just like in those movies, the real danger isn’t always in the technology itself—it’s in how well (or poorly) it’s

implemented. So, while you may never need to worry about someone stealing your eyeball, you should definitely worry about weak encryption, poor anti-spoofing measures, and bad implementation.

15.1.6 Exercises

1. How does biometric authentication work?
2. What is stored during biometric registration?
3. What are spoofing and presentation attacks?
4. Why is storing biometric templates insecurely a problem?

15.2 Authenticating using hardware keys

Alice had struggled with every frustrating login method imaginable - passwords she could never remember, OTPs that expired too soon, and magic links that always ended up in spam. But today, her boss handed her a tiny USB-like device and said, "This is your new login method. Keep it safe."

Alice examined it skeptically. This little thing? It didn't have a screen, a password, or even a blinking light. How was this supposed to keep her accounts secure? But when she plugged it into her laptop and tapped the button, magic - she was in. No passwords, no codes...

It reminded her of Lara Croft unlocking an ancient tomb with a mystical key —except instead of opening a chamber filled with priceless artifacts, Alice had just accessed her email. This little device was her "Triangle of Light"—the key that granted instant access while keeping digital treasure (a.k.a. her data) locked away from intruders. No phishing, no stolen passwords—just a solid, hacker-proof authentication method.

But how does this real-world digital relic actually work? And what happens if Alice loses it (because, let's be real, she probably will – she can't even keep a pair of glasses for more than a couple of months)?

Think of a hardware security key as a super-smart, uncopiable key that only fits your digital locks. Instead of typing passwords or waiting for a one-time code, you simply plug in (or tap) your key, which proves to the system that

it's really you without exposing any secret information that hackers could steal.

1. *You try to log in* – You go to your email, bank, or any account that supports hardware keys. Instead of asking for a password, the system says, "Hey, prove you're the real owner!"
2. *Your hardware key responds* – You plug in or tap your key (via USB, NFC, or Bluetooth). The key doesn't just send a stored password—it uses a unique, cryptographic handshake to prove it's legit.
3. *The system verifies the key* – Your account checks the key's response, confirming it's really yours and not a fake or a hacker trying to trick the system.

When a user registers a hardware security key with an online service, the device generates a unique key pair:

- Private Key (Stored Securely on the Device) – This key never leaves the security key and is used to sign authentication requests.
- Public Key (Stored by the application that requires it for authentication) – This key is registered with the website or app and is used to verify authentication attempts.

Note

The private key is never exposed or transmitted, making it impossible for attackers to steal or reuse it.

When a user tries to log in using a hardware key, the process follows a challenge-response model, ensuring that no sensitive data is sent over the network (figure 15.3):

1. The website sends a challenge – Instead of asking for a password, the website generates a random challenge (nonce) and sends it to the security key.
2. The security key signs the challenge – The key digitally signs the challenge using its private key and returns the signed response to the website.
3. The website verifies the signature – The server checks the response

against the registered public key. If it matches, authentication is successful.

Figure 15.3 The authentication process using a hardware key. The app generates a random value and asks the key to sign it with its private key. The app then verifies the signature using its public key. The public key from the pair needs to already be configured at the app level. This configuration happens during the registration step.

The user

Acme. Inc.

Hardware key



1. User requests to authenticate.

2. The app generates a random value and asks the hardware key to sign this value with its private cryptographic key.

3. The key sends back the signed value.

4. The app validates the signature using the public key and authenticates the user.

Another important thing is that hardware security keys use an origin/domain binding technique, meaning they will only work on the site they were registered with. If you register your key with <https://securebank.com>, it will only sign authentication requests for that exact domain. If a phishing site (<https://fakebank.com>) tries to authenticate, the security key won't sign the request, preventing credential theft. This makes hardware keys immune to phishing attacks, unlike passwords and OTPs.

After everything we've covered, one thing is clear—hardware security keys are like digital vault keys that hackers can't steal, phish, or guess. Unlike passwords that get leaked, OTPs that arrive too late, and biometrics that can't be changed once compromised, these tiny devices offer rock-solid security with a simple tap.

Sure, they might seem old-school compared to facial recognition or magic links, but hardware keys outperform them all when it comes to keeping hackers out. No phishing attack, no man-in-the-middle interception, no sneaky deepfake can trick a security key into authenticating on the wrong website.

Of course, no system is perfect—losing a hardware key can be a hassle, and not every website supports them (yet). But as more services embrace passwordless authentication, hardware keys are proving to be the gold standard of security.

15.2.1 Exercises

5. What is a hardware security key?
6. How does a hardware key prevent phishing?
7. What is the attestation object in WebAuthn?
8. Why are hardware keys considered more secure than biometrics?
9. What happens if you lose your hardware key?

15.3 Implementing WebAuthn authentication

In sections 15.1 and 15.2, we dissected WebAuthn definitions, exploring the

advantages and disadvantages of different approaches. Now, it's time to delve into code and review the practical aspects of implementation. In this section, we'll work through a simple example to demonstrate how to implement authentication with WebAuthn.

We'll begin with the registration process. For this purpose, we'll create a basic web page using plain JavaScript, allowing users to set their authentication credentials. Additionally, we'll develop a demo backend to support this functionality.

The second step is authentication. We'll implement another simple web page using plain JavaScript, where users will be prompted to authenticate.

Listing 15.1 illustrates the first step in creating a simple registration web page. The page contains only a basic button that, when clicked, triggers a simple JavaScript function. This function—which we will develop in the upcoming listings—is responsible for gathering the necessary registration details.

Listing 15.1 Preparing the Javascript registration function

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>WebAuthn Registration</title>
</head>
<body>
    <h2>WebAuthn Registration</h2>
    <button onclick="registerWebAuthn()"> #A
        Register WebAuthn Credential
    </button>

    <script>
        async function registerWebAuthn() { #B
            // to implement
        }
    </script>
</body>
</html>
```

Listing 15.2 begins the implementation of the JavaScript registration function. We start by checking whether the current system supports WebAuthn hardware-based authentication for registration. Since the web page runs in a browser, this check can be performed easily using standard JavaScript, as shown in the listing.

Listing 15.2 Checking if the browser supports WebAuthn

```
async function registerWebAuthn() {
    if (!window.PublicKeyCredential) { #A
        console.error("WebAuthn is not supported in this browser.");
        return;
    }
    // to continue implementation
}
```

Listing 15.3 adds the logic for retrieving the public key credentials. These credentials must be stored on the server side, as they are essential for verifying responses from the hardware authentication device. The retrieval process is performed only if the system supports hardware authentication. Otherwise, an error message is logged to the console, and the registration process is halted.

This simplified approach is intended for learning purposes, allowing you to focus solely on the core steps. Naturally, a more polished user experience—with proper error handling and user feedback—would be necessary in a real-world application.

Listing 15.3 Request for obtaining the public key credentials for registration

```
async function registerWebAuthn() {
    if (!window.PublicKeyCredential) {
        console.error("WebAuthn is not supported in this browser.");
        return;
    }

    const publicKeyCredentialCreationOptions = {
        challenge: new Uint8Array(32), #A
        rp: { name: "My WebAuthn App" }, #B
        user: { #C
            id: new Uint8Array(16), #C
        }
    }
}
```

```

        name: "user@example.com", #C
        displayName: "User Example" #C
    },
    publicKeyCredParams: [{ type: "public-key", alg: -7 }],
    authenticatorSelection: { authenticatorAttachment: "platform"
        timeout: 60000, #D
        attestation: "direct"
    };
}

// to continue implementation
}

```

Listing 15.4 completes the process by creating the public key using the previously retrieved details. If the operation is successful, the key's essential components—such as the credential ID, the key type, the algorithm, and the raw public key—are printed to the console; otherwise, an error message is logged.

In a real-world application, this public key object would be sent to the server, typically as part of a JSON payload via an HTTPS POST request. The server stores the credential ID and associated public key, which it later uses to verify signed authentication responses during login attempts.

For demonstration purposes, we simply log the key details to the console to validate that the front-end portion of the WebAuthn registration flow is functioning as expected.

Listing 15.4 Requesting the public key creation

```

async function registerWebAuthn() {
    if (!window.PublicKeyCredential) {
        console.error("WebAuthn is not supported in this browser.");
        return;
    }

    const publicKeyCredentialCreationOptions = {
        challenge: new Uint8Array(32),
        rp: { name: "My WebAuthn App" },
        user: {
            id: new Uint8Array(16),
            name: "user@example.com",
            displayName: "User Example"
        },

```

```

pubKeyCredParams: [{ type: "public-key", alg: -7 }],
authenticatorSelection: { authenticatorAttachment: "platform"
timeout: 60000, // Timeout in milliseconds
attestation: "direct"
};

try {
  const credential = await navigator.credentials
➥.create({ #A
    publicKey: publicKeyCredentialCreationOptions
});

if (!credential) { #B
  console.error("Credential creation failed.");
  return;
}

const attestationObject = #C
  btoa(String.fromCharCode(...new
    Uint8Array(credential.response.attestationObject)));
const clientDataJSON =
  btoa(String.fromCharCode(...new
    Uint8Array(credential.response.clientDataJSON)));
const credentialId =
  btoa(String.fromCharCode(...new
    Uint8Array(credential.rawId)));

console.log("WebAuthn Registration Data:");
console.log("Attestation Object:", attestationObject);
console.log("Client Data JSON:", clientDataJSON);
console.log("Credential ID:", credentialId);
} catch (error) {
  console.error("WebAuthn registration failed:", error);
}
}

```

Now it's time to run the page. Since it's just an HTML file, your first instinct might be to double-click it and let the browser do its thing. Unfortunately, WebAuthn isn't that easygoing—it insists on running in a secure context. Simply opening the file directly won't work.

But don't worry, you don't need to buy a domain or spin up some heavy enterprise server just to impress WebAuthn. Thankfully, localhost is

considered a “real” domain for security purposes, so any local web server will do the trick—even the small-but-mighty kind.

Since this book focuses on Java, we’ll stick with the tools Java offers. Starting with Java 21, the JDK includes a handy little HTTP server you can launch right from the command line:

```
jwebserver --port 8000
```

If you use the command shown in the previous snippet, the web page file must be located in the current working directory (.) from which you start the web server. Alternatively, you can specify a different folder by using the --directory option, like this:

```
jwebserver --port 8000 --directory /path/to/your/files
```

Figure 15.4 shows you the usage of the `jwebserver` command for a Java 21 installation.

Figure 15.4 Running a local web server using the `jwebserver` command from the JDK. The server is started in the `bin` directory of the JDK installation, binding to 127.0.0.1 on port 8000. This allows the HTML page to be served over a secure context (localhost), which is required for WebAuthn functionality.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

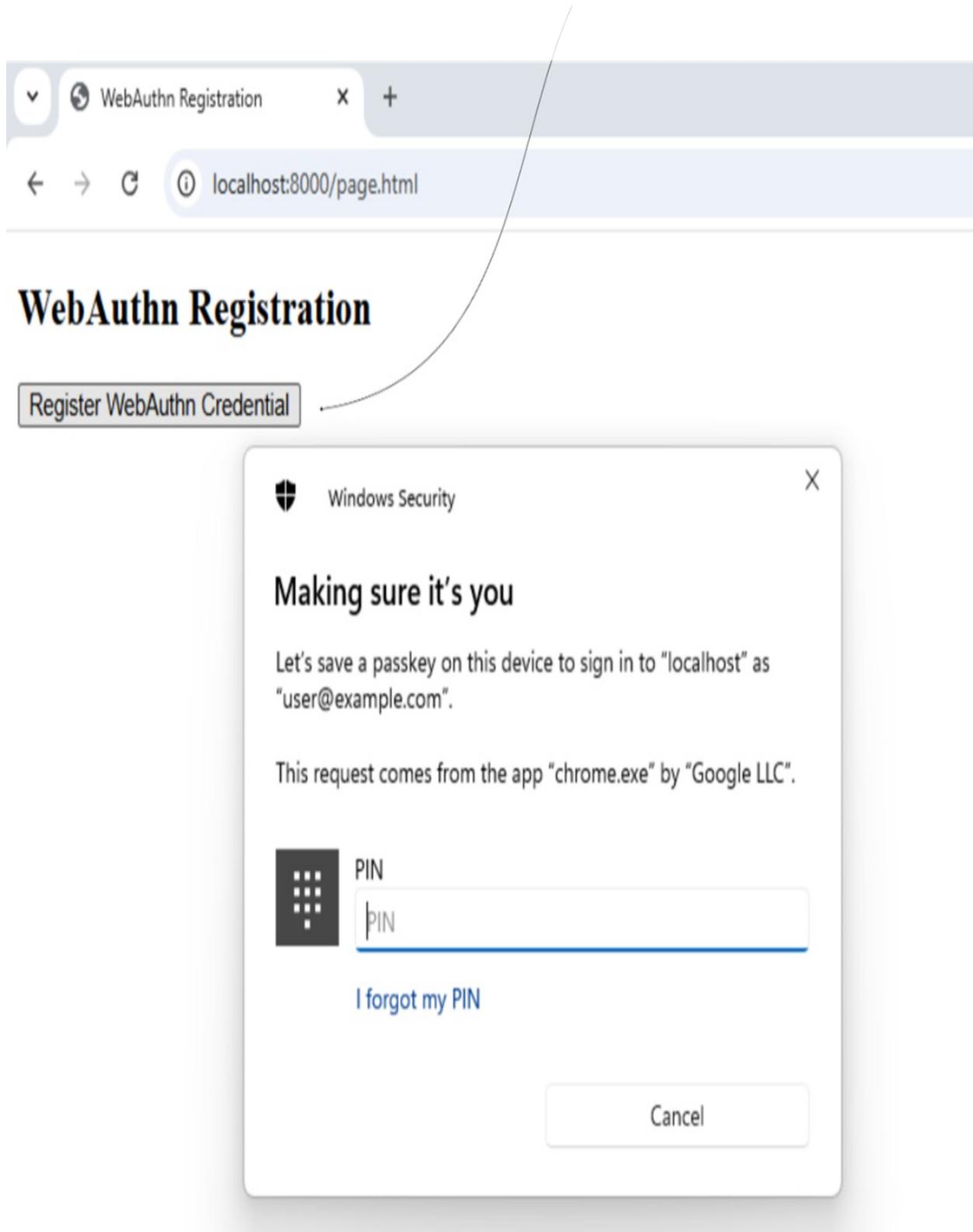
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\lspilca\.jdks\azul-21\bin> ./jwebserver --port 8000
Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".
Serving C:\Users\lspilca\.jdks\azul-21\bin and subdirectories on 127.0.0.1 port 8000
URL http://127.0.0.1:8000/
```

Accessing the page on localhost and port 8000 will display the page. Click on the button to initiate the registration process as shown in figure 15.5.

Figure 15.5 When the "Register WebAuthn Credential" button is clicked, the browser initiates the WebAuthn registration process. The system prompts the user with a Windows Security dialog to confirm their identity and save a passkey for the specified domain (localhost in this case), using platform authentication.

When clicking on the button, the dialog box appears to request the registration.



The prompt shown during WebAuthn registration can vary depending on the device, browser, and authentication method available. In the example shown, the system prompts for a PIN because it is using Windows Hello as the platform authenticator, and the current configuration relies on PIN-based verification.

Windows Hello supports several user verification methods, including PINs, fingerprints, and facial recognition. If the device has biometric hardware (like a fingerprint sensor) and it's configured, the system may prompt the user to authenticate using that method instead of a PIN. This ensures that only the legitimate user can register a passkey on the device.

On other platforms, the experience differs. For example, macOS users might be prompted to use Touch ID, while Android users may see a native biometric prompt for fingerprint or facial recognition. If the user is registering with an external security key (such as a YubiKey), the browser may prompt them to insert or tap the device and, if required, enter a PIN associated with that hardware key.

In short, while the underlying WebAuthn process remains the same, the user experience is platform-dependent. Each system chooses the most appropriate verification method based on its configuration and available hardware.

Figure 15.6 shows the output in the browser's developer console, confirming the completion of the front-end portion of the WebAuthn registration process.

Figure 15.6 Output in the browser's developer console after triggering the WebAuthn registration process. The console displays the attestation object and client data in base64-encoded form, representing the public key credential generated by the browser.

The console shows the public key details that the app printed.

The screenshot shows a browser window titled "WebAuthn Registration" at "localhost:8000/page.html". The console tab is selected, displaying several configuration options:

- Hide network
- Preserve log
- Selected context only
- Group similar messages in console
- Show CORS errors in console

A yellow warning message is present:

⚠️ ► publicKey.pubKeyCredParams is missing at least one of the default algorithm(s).
<https://chromium.googlesource.com/chromium/src/+main/content/browser/weba>

The "WebAuthn Registration Data:" section contains a large, illegible string of base64 encoded data representing the registration object.

Below the registration data, two more sections are shown:

Client Data JSON: eyJ0eXAiOi...
Credential ID: 639AchgS...
...
...
...

Listing 15.5 illustrates the backend component responsible for handling the WebAuthn registration request. In this stage of the flow, the backend's role is relatively simple: it receives the credential data from the front end and stores it, associating it with a specific user account. This typically includes values such as the credential ID, the public key, and optionally the client data.

While the example shown here uses a placeholder response and does not persist the data, in a real-world application you would store these details in a database, linked to a user record. For instance, if the user is already logged in or identified during the registration flow, you could associate the credential with their email address or username: email/username -> [credentialId, publicKey, algorithm]. This stored data will later be used to validate authentication attempts, ensuring that only users with a valid credential can sign in successfully.

Listing 15.5 The backend side stores the registration details associated with an account

```
@RestController
@RequestMapping("/api/webauthn")
public class WebAuthnController {

    @PostMapping("/register")
    public String register(@RequestBody RegistrationRequest request
    #A
        return "Registration successful";
    }
}
```

OK. Let's now talk about authentication which implies two steps:

1. The backend generates a challenge (random value)
2. The client needs to sign the backend's challenge and send the signed value to the backend.
3. The backend verifies the signature using the public key stored during the registration.

We will assume that the registration already took place, since the system needs to know the user and have a public key registered for its hardware device to allow them authenticate.

The backend needs to provide two endpoints:

- And endpoint to provide the client with the challenge (step 1)
- And endpoint to validate the data sent by the client. (step 3).

As with the previous example, we've intentionally kept things as simple as possible to help you more easily digest the explanation. Including a full, production-ready implementation in a book would be impractical, but the code and concepts presented here should give you a solid foundation to build upon. When you're ready to implement this in a real-world scenario, a good understanding of your chosen framework is essential. If you're working with Spring and need a refresher, I highly recommend *Spring in Action* by Craig Walls and *Spring Security in Action* by Laurentiu Spilca - both are excellent resources for deepening your knowledge.

Listing 15.6 presents a simplified implementation of the endpoint that provides the client with a challenge required for authentication. The application identifies the user based on the provided username and generates a secure, random challenge. It then stores an association between the user ID and the challenge in a temporary store.

When implementing this in a real-world scenario, pay close attention to how and where this association is stored. If your application needs to be horizontally scalable—running across multiple instances - storing this data in memory (e.g., using a Map) is unsuitable. Instead, consider using a shared storage mechanism such as a database or a distributed cache (e.g., Redis) to ensure node consistency and availability.

The application then prepares the response and returns the challenge to the client. In addition to the challenge itself, notice that I've included the previously stored credential ID in the response. This is done by populating the `allowCredentials` field, which explicitly tells the client which credentials are permitted for authentication.

Providing `allowCredentials` is highly recommended when you already know the user's identity (for example, after they've entered a username or email address). By doing so, you're instructing the browser to limit the authentication options to a specific credential or set of credentials that are

registered for that user. This not only improves performance but also enhances security and user experience by avoiding unnecessary prompts or confusion.

If you omit the `allowCredentials` field, the browser assumes you're performing username-less login and may prompt the user to select a credential from any passkey or security key available on their device. This can lead to unintended consequences. For example, if a user has multiple credentials registered for different services or identities, they may accidentally attempt to authenticate with the wrong one—causing the authentication to fail or produce misleading errors.

Listing 15.6 A demonstration of the challenge endpoint

```
@PostMapping("/authenticate/challenge")
public Map<String, Object> getAuthenticationChallenge(
    @RequestBody Map<String, String> request) {
    String username = request.get("username");

    String storedCredentialId = getUser(username); #A

    byte[] challenge = new byte[32]; #B
    new SecureRandom().nextBytes(challenge);

    store.put(storedCredentialId, challenge); #C

    Map<String, Object> allowCredential
    => new HashMap<>(); #D
    allowCredential.put("type", "public-key");
    allowCredential.put("id", storedCredentialId);

    var encChallange = Base64.getEncoder()
    =>.encodeToString(challenge); #E

    Map<String, Object> response = new HashMap<>(); #F
    response.put("challenge", encChallange );
    response.put("allowCredentials", List.of(allowCredential));

    return response;
}
```

Beyond simply verifying the signature, the server can and should enforce several additional rules to strengthen the security of the authentication

process:

- *Challenge timeout*: The server should associate each challenge with a limited time window - typically a few minutes - during which the user must complete authentication. This prevents an attacker from capturing and reusing an old challenge at a later time. After the timeout expires, the server should reject any authentication attempt using that challenge. Implementing this mechanism significantly reduces the risk of replay attacks or unauthorized use of stale challenges.
- *Relying Party ID enforcement*: The server should verify that the authentication data's Relying Party ID (`rpId`) matches the expected domain (e.g., `example.com`). This ensures that the client genuinely interacts with your server and is not tricked into authenticating against a malicious domain. Enforcing the correct `rpId` protects against phishing and man-in-the-middle attacks where a rogue site attempts to hijack the authentication process.

Listing 15.7 demonstrates a basic implementation of the authentication logic. This example is intentionally simplified to help you understand the key steps involved in the process. In a production-grade application, especially if you're using Spring, this logic should be properly integrated into the authentication mechanism using Spring Security's standard design patterns. If you need a refresher on how to do that, the best resource is *Spring Security in Action* by Laurentiu Spilca.

In this listing, you can see how the server receives the authentication details from the client. The most essential elements are the credential ID and the signature generated by the authenticator for the previously issued random challenge. The backend uses the public key, which was stored during registration, to verify the signature cryptographically. If the verification is successful and the signature matches the challenge, the authentication is considered successful, and the user can be granted access.

Listing 15.7 The authentication endpoint

```
@PostMapping("/authenticate")
public String authenticate(
    @RequestBody AuthenticationResponse response) {
```

```

String credentialId = response.getId(); #A
byte[] signature = Base64.getDecoder()
    .decode(response.getResponse().getSignature());
byte[] authenticatorData = Base64.getDecoder()
    .decode(response.getResponse().getAuthenticatorData());
byte[] clientDataJSON = Base64.getDecoder()
    .decode(response.getResponse().getClientDataJSON());

byte[] storedPublicKey
➥= getRegisteredPublicKey(credentialId); #B
byte[] originalChallenge
➥= store.get(credentialId); #B

var success = verifySignatureAndAuthenticate( #C
    storedPublicKey,
    originalChallenge,
    signature);

if (success) return "Authentication successful";
else throw new UnsuccessfulAuthenticationException();
}

```

Note

The signature verification might be challenging to implement on your own. Also, implementing it yourself could lead to errors, which in turn could introduce vulnerabilities. I recommend you use instead a known library for implementing this capability such as WebAuthn4J (<https://github.com/webauthn4j/webauthn4j>) for Java apps.

And there you have it—the core steps of WebAuthn authentication from both the frontend and backend sides. While we've kept things intentionally simple to help you grasp the essentials, you now have all the key pieces to move toward a secure, passwordless login flow.

Of course, in a real application, you'll want to tighten things up: use proper cryptographic verification, secure your challenge storage, and integrate cleanly with your authentication framework. But for now, you've earned the right to say "Yeah, I know how WebAuthn works." Just don't try it in production with an in-memory map and a prayer.

Now, take a deep breath—you just tackled a modern authentication flow that combines cryptography, browser APIs, and backend logic.

15.3.1 Exercises

10. What are the main steps of registration in WebAuthn?
11. Why does WebAuthn require localhost or HTTPS?
12. What's the role of the challenge during authentication?
13. Why is it important to verify the challenge and origin?
14. Create a basic WebAuthn registration and authentication demo using JavaScript and a simple backend:

Part A: WebAuthn Registration (Frontend)

Requirements:

- A simple HTML page with a “Register Credential” button.
- JavaScript code that:
 - Checks WebAuthn support.
 - Generates a credential using `navigator.credentials.create()`.
 - Logs or displays: credential ID, client data JSON

Bonus: Convert the data to Base64 and show it as a simulated payload to send to the backend.

Part B: Simulate the Backend

- Use Spring Boot or a minimal Java HTTP server to:
 - Accept POST requests at `/api/webauthn/register`
 - Log and “associate” credentials with users in-memory (e.g., `Map<String, WebAuthnCredential>`)

Part C: Authentication Simulation

Implement:

- A challenge generator (32-byte random value).
- A frontend that:

- Requests this challenge from the backend.
- Signs it using `navigator.credentials.get()`
- Sends the response back to the backend for verification.

15.4 Answers to exercises

1. How does biometric authentication work?

It verifies identity using traits like your fingerprint, face, or voice. These are scanned and matched with data stored securely on your device.

2. What is stored during biometric registration?

Not the actual fingerprint or face. A math-based template of your biometric data is stored in a secure chip like the TPM or Secure Enclave.

3. What are spoofing and presentation attacks?

Spoofing tricks sensors using fake fingerprints or 3D masks.

Presentation attacks use photos, videos, or AI-generated data to fool systems.

4. Why is storing biometric templates insecurely a problem?

Because unlike passwords, biometrics can't be changed. If stolen, attackers can impersonate users forever.

5. What is a hardware security key?

It's a physical device that stores a private key and uses it to prove your identity during login, without ever sending the key over the network.

6. How does a hardware key prevent phishing?

It's tied to the original website (origin binding), so it won't respond to fake site, even if they look identical.

7. What is the attestation object in WebAuthn?

It's data the browser sends during registration that proves a real, trusted device created the key pair.

8. Why are hardware keys considered more secure than biometrics?

Because they use cryptography, never expose secrets, and can't be faked or copied like biometric traits might be.

9. What happens if you lose your hardware key?

You're locked out unless you registered a backup key. That's why having a backup is best practice.

10. What are the main steps of registration in WebAuthn?

The browser checks support, creates a key pair, and sends the public key to the backend. The private key stays safely on the device.

11. Why does WebAuthn require localhost or HTTPS?
Because secure context is mandatory, it protects the key and data from interception during registration and login.
12. What's the role of the challenge during authentication?
It's a random string the server sends. The device signs it with its private key to prove it's really the same device.
13. Why is it important to verify the challenge and origin?
To prevent replay and phishing attacks. Only the original challenge from the right domain should be accepted.
14. Create a basic WebAuthn registration and authentication demo using JavaScript and a simple backend.
URL to solution:
https://github.com/Software-Security-For-Developers/software-security-for-developers/tree/main/ssfd_ch15_exercise14

15.5 Summary

- Passwordless authentication improves security and convenience by eliminating weak or reused passwords.
- Spoofing attacks (deepfakes, fake fingerprints) and stolen biometric templates pose risks, such as unauthorized access to sensitive accounts, bypassing multi-factor authentication, and long-term identity compromise
- Storing biometric data in secure hardware (TPM, Secure Enclave) and implementing anti-spoofing measures improves security.
- Hardware security keys provide the strongest authentication by using public-key cryptography.
- Hardware security keys eliminate phishing risks but require physical possession, making backup keys necessary.
- Using FIDO2/WebAuthn and enforcing domain binding ensures maximum security.
- Passwordless authentication reduces phishing, credential leaks, and login friction.
- No single method fits all use cases. Choosing the right approach depends on security needs, usability, and risk factors.
- Proper implementation is key—even the most secure authentication method can be exploited if configured poorly.

- Following best practices, encryption standards, and secure handling of authentication data ensures a stronger, safer login experience.

Part 5: Securing service-to-service call chain

We've reached the final part of this book. Up to now, you've learned how to use cryptography, build trust with certificates and TLS, and authenticate users with modern identity protocols. But securing applications doesn't stop at users. In large systems, the real challenge is keeping the services themselves honest — proving who they are, and making sure they can only do what they're allowed to do.

In this part, we'll look at service identity (chapter 16), the foundation for secure service-to-service communication in a zero-trust world. Then we'll turn to authorization at scale (chapter 17), exploring RBAC, ABAC, and ReBAC — and how to choose the right model for your architecture without drowning in complexity.

This closing part ties everything together. By the end, you'll have the tools to secure not just users, but the services and workflows that make up modern cloud-native systems. It's a fitting way to conclude our journey: from understanding the math behind cryptography all the way to building systems that stay secure at scale.

16 Implementing service identity

This chapter covers

- Understanding what service identity is
- Enforcing service-to-service authorization directly within the application code
- Exploring infrastructure-level approaches to delegate identity enforcement
- Choosing between application and infrastructure-level strategies

If you've ever tried to debug why one service is refusing to talk to another, only to discover it's because the caller was impersonating someone else, you already understand why service identity matters. In a world where machines talk to machines more than people talk to people, knowing who's calling isn't just a nice-to-have.

The days when we could trust everything inside the perimeter are gone. The perimeter is gone. The services are multiplying a lot, and every call in your system should be treated with suspicion. Service identity is to bring back order in the chaos. It's the badge each service wears to say, "Yes, it's really me, and here's the cryptographic proof."

16.1 What service identity is

Service identity is the mechanism that allows systems to determine who a service (app) is. Just like user identity helps us determine who a human is, service identity does that for apps in a system. Without a reliable way to identify services, authorization, auditing, and secure communication become guesswork.

If you can't prove who's making a request, then granting access is a gamble, and every decision could be a security breach in disguise. Without a strong service identity, you accept anonymous API calls and hope they're from a

friend. And hackers love optimism.

Definition

Service identity is a unique, verifiable identity assigned to a service instance or application component.

I'll tell you now a story about Alexey, a seasoned hacker. Alexey found a breach, an exposed debug pod inside a company's Kubernetes cluster. There was no password, no protection. It was like someone had left the back door wide open (yes, this should never happen, but sometimes it does, and that's why the Zero-trust principle is essential).

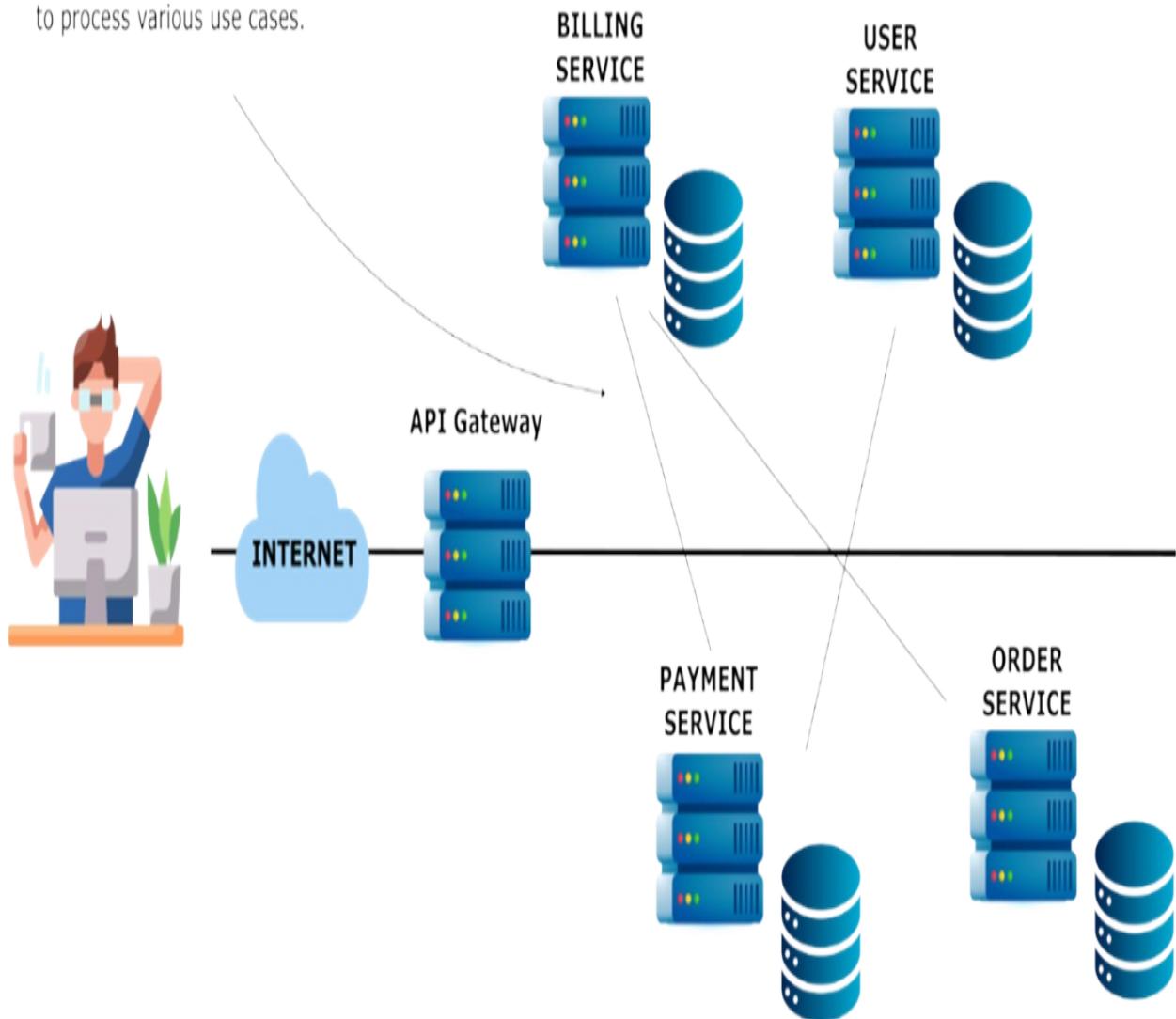
Definition

Zero Trust is a security principle that means never automatically trusting anything, whether it comes from inside or outside your system. Instead, everything must be verified: every user, every device, and every service request.

Once inside, the hacker began to explore. Many services were talking to each other, like order, billing, and user service. All of them trusted each other just because they were inside the same network. No one checked who was really sending the requests (figure 16.1).

Figure 16.1 In a system multiple services communicate with one another to solve various use cases. If the communication isn't secure, attackers may manage to exploit vulnerabilities and change or steal data.

Services communicate with one another to process various use cases.



Alexey focused on the billing service. It accepted requests from the order service to process invoices. But instead of checking for a digital signature or a certificate, it just looked at a simple header:

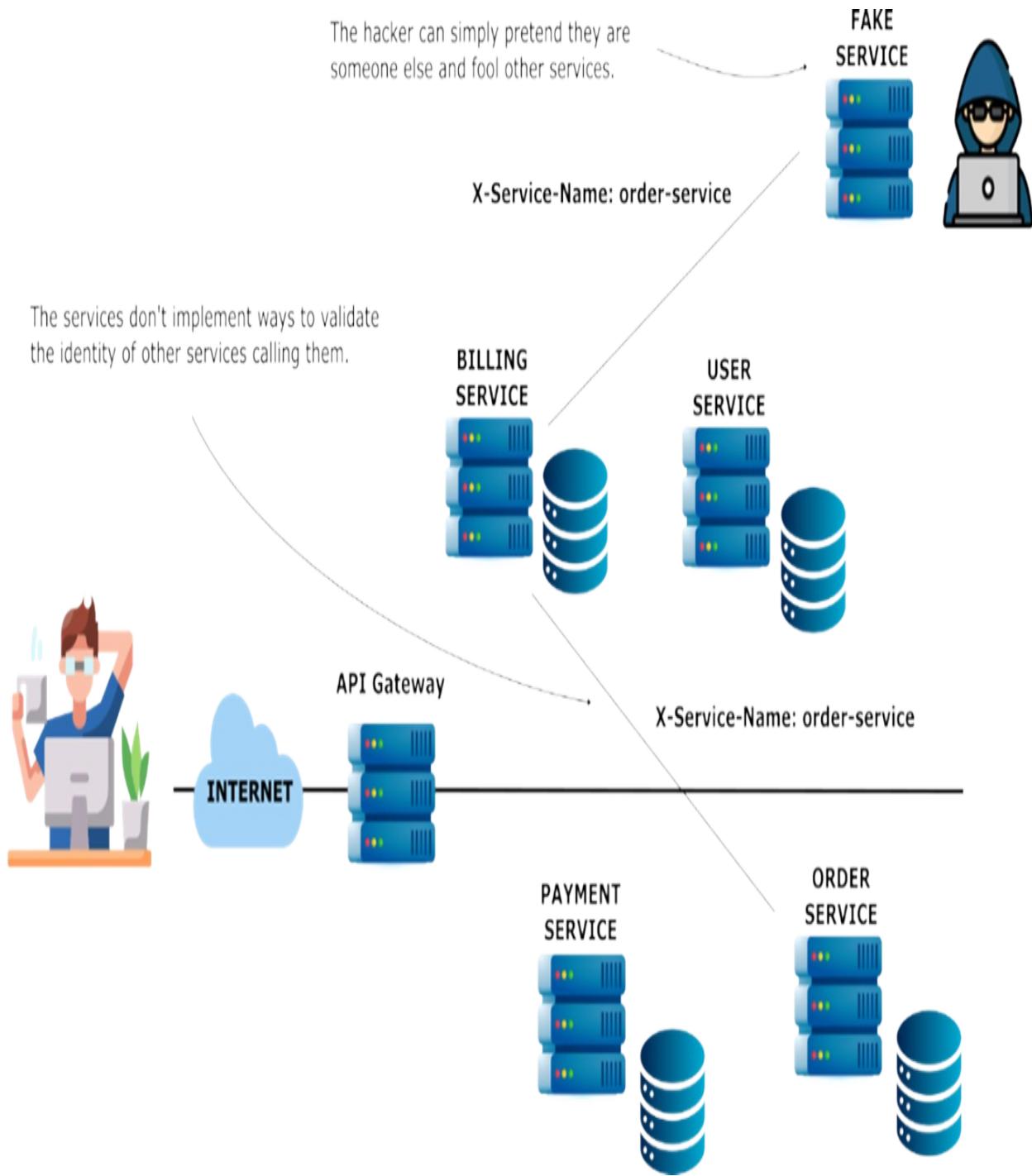
X-Service-Name: order-service

“That’s it? No secret key, no certificate, nothing to prove who’s sending the request? Perfect.”, he thought.

So he created his own service (figure 16.2). Then he started sending fake requests to the billing service, adding the header that claimed to be from the

order service. It worked! The billing service couldn't tell the difference. It accepted fake requests, processed fake invoices, and even leaked real billing data.

Figure 16.2 Services that trust incoming messages without verifying the sender's identity are vulnerable—if an attacker gains access, they can impersonate trusted services and send messages to manipulate or extract data from the system.



The hacker kept going quietly, generating fake transactions, modifying amounts, and collecting information. All because the system had no reliable way to check service identity.

This kind of attack is possible only when services can't prove who they are. Let's look at the building blocks that help prevent this. We'll take a look at

the key components that help keeping our system more secure from the service communication point of view:

- *Identity*: Typically encoded in certificates, tokens, or metadata
- *Authentication*: Verifying the identity (e.g. using mTLS, JWTs, signatures)
- *Trust Anchors*: Who vouches for this identity? (e.g. certificate authority, token issuer)
- *Scope of identity*: What the identity is assigned to and how specific it is.

Every service in a system needs a way to say, “This is who I am.” That’s what we call *identity*. Just like people carry passports or ID cards, services also have something that proves who they are. This identity usually comes in the form of a certificate, a token (like a signed JWT), or some trusted metadata provided by the system, such as a Kubernetes service account. These are unique to each service and help other parts of the system recognize them. Without this kind of identity, any service could claim to be anything, and that’s when bad things can happen.

Once a service says, “This is who I am” (*identity*), the next step is to check if that’s true. That’s called *authentication*. It’s how a system ensures the service owns the identity it claims. This is usually done using tools like mutual TLS (mTLS), where both services show and check certificates, or with JWT tokens, where the token is signed by someone trusted. Think of it like checking the hologram and signature on a passport, you’re not just reading the name, you’re proving it’s not fake.

But who decides which identities are real? That’s where *trust anchors* come in. A trust anchor is a system or authority that everyone agrees to trust. It could be a certificate authority (CA) that signs service certificates, or a token issuer that creates signed JWTs. If a service shows a certificate signed by a trusted CA, others will believe it. If it uses a token from a known and trusted source, that’s accepted too. It’s like a friend you trust saying, “Yes, this person is who they say they are.” Without trust anchors, services wouldn’t know who to believe.

Finally, we need to consider the *scope of identity*. The scope of identity means how specific the identity of a service is. Sometimes, all copies of a

service share the same identity, like giving the same badge to every delivery driver. That's a wide scope. Other times, each service instance or container gets its own unique identity, like giving a personal badge to each driver, with their name and time of access. That's a narrow scope. Using a narrow scope makes your system more secure because it lets you control access more precisely and see exactly who did what. The more dynamic your system is, the more useful it is to have a smaller, more specific scope for each service identity.

Now that we understand what service identity is and why it matters, the next step is learning how to use it to control access between services. In the next section, we'll look at how you can implement service authorization directly in your application code, verifying who's calling and deciding what they're allowed to do. Then, in section 16.3, we'll explore how to push that responsibility down into the infrastructure, using tools like service meshes and gateways to enforce security without changing your application logic. Both approaches have their strengths, and knowing when to use each one is key to building secure, maintainable systems.

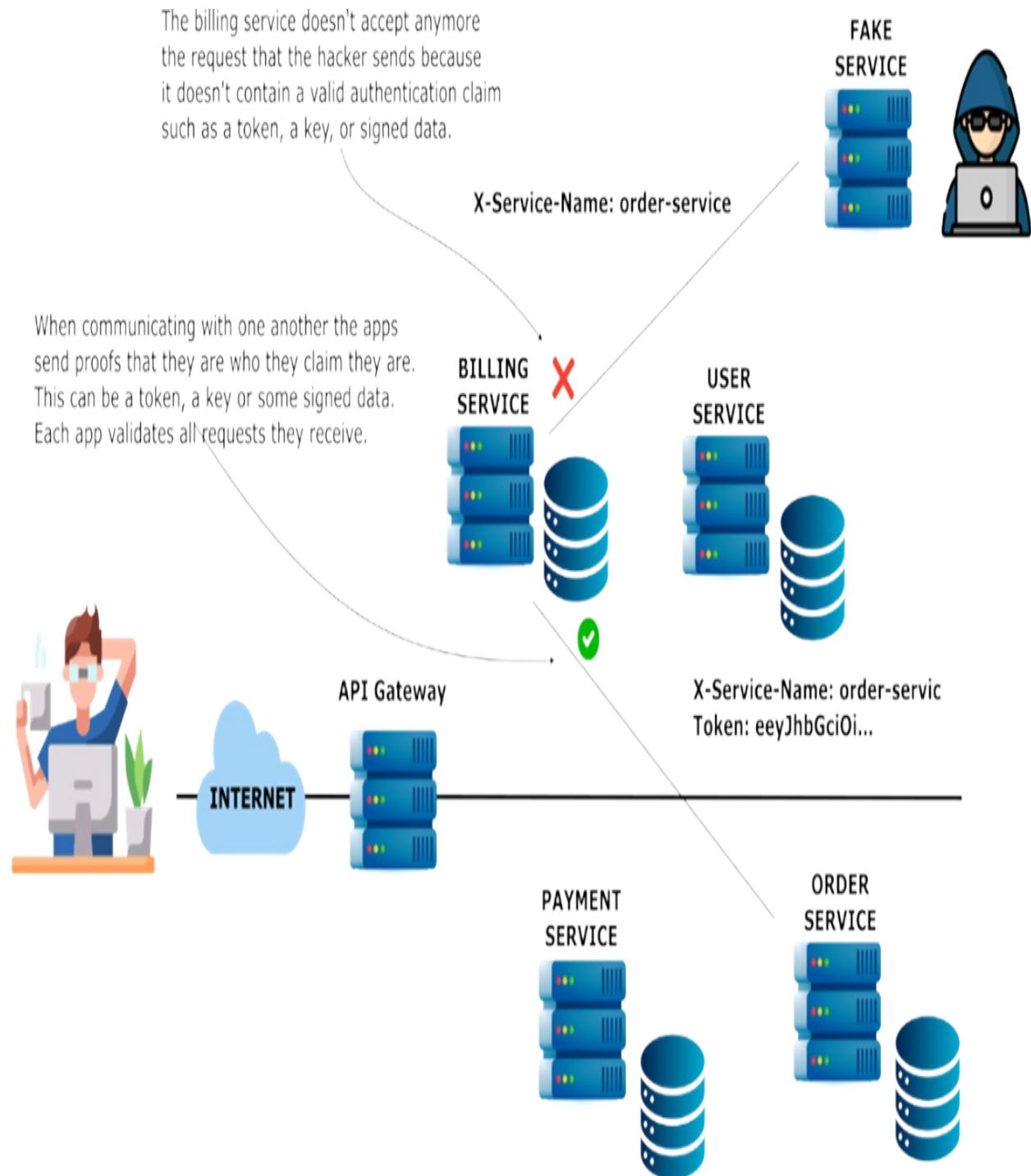
16.1.1 Exercises

1. What is service identity and why does it matter?
2. What's the difference between identity and authentication?
3. What's a trust anchor?
4. Why does Zero Trust apply here?

16.2 Implementing service authorization at the application level

Once a service knows who is calling (through authentication), the next question is: Is this caller allowed to do what they're trying to do? That's where authorization comes in. "At the application level" means the service itself, through code or middleware, makes the decision. It checks the caller's identity and applies rules like: "Can order-service call /process-invoice on billing-service?" If the answer is yes, the request goes through. If not, it gets blocked right there in the application (figure 16.3).

Figure 16.3 Each application includes logic to validate incoming requests. Every request should contain a token, key, or other proof that identifies the sender as a trusted counterpart. If the request lacks valid proof, the service rejects it.



This approach gives you fine-grained control, because you can build logic directly around your business needs. But it also means every service has to

play by the rules, and if one forgets (or gets lazy), you might end up with gaps in your defenses. Using application-level authorization means that each service is responsible for checking who is making a request and whether that request should be allowed. This gives you fine-grained control, because you can write logic that is very specific to your business. For example, a service can say, “Only the order-service is allowed to create invoices,” or “Only users with the role manager can approve refunds over \$1,000.”

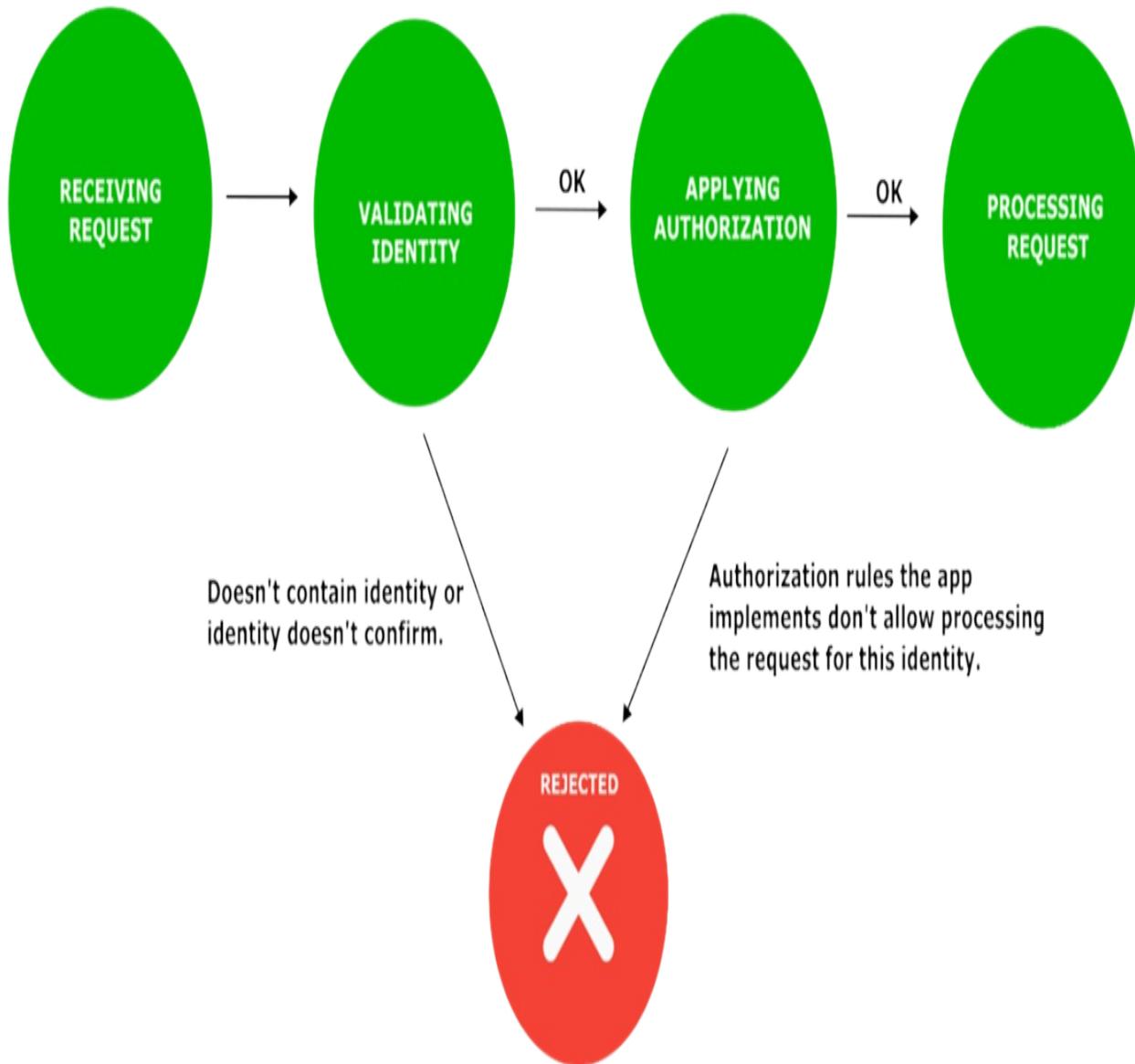
This flexibility is powerful, especially when the decision depends on something only the application knows such as user roles, order history, or product inventory. You can tailor the rules exactly to fit your needs, and enforce them directly inside the part of the code that handles the request.

But there’s a downside: because each service is in charge of enforcing its rules, you must trust every team to get it right. One service might perform proper checks, while another might skip a step or apply weaker rules. This can lead to inconsistent policies across the system and create security holes that attackers can exploit.

If we are to describe service authorization at the app level in a few simple steps they are (figure 16.4):

1. The caller (another service) sends a request along with a token or credential that proves who it is.
2. The receiving service (app itself) validates that token, checking that it’s real and unexpired.
3. The service (app itself) checks a set of authorization rules based on the caller’s identity.
4. If everything checks out, the request is processed. If not, it’s denied.

Figure 16.4 Steps the app follows for validating a request.



What tools can you use to implement service authorization at the app level? To implement service authorization at the application level, most systems rely on one of a few common patterns for identifying and verifying callers. Each comes with trade-offs in terms of security, complexity, and flexibility:

1. *The OAuth2 client credentials grant type* - A service authenticates with an authorization server to obtain a token, which it uses to prove its identity when calling other services.
2. *API keys* - A simple static token included in requests to identify the caller, often easy to use but less secure and harder to manage at scale.
3. *Custom authorization logic* - Application-specific rules written in code

to decide whether a request is allowed, offering flexibility but requiring careful and consistent implementation.

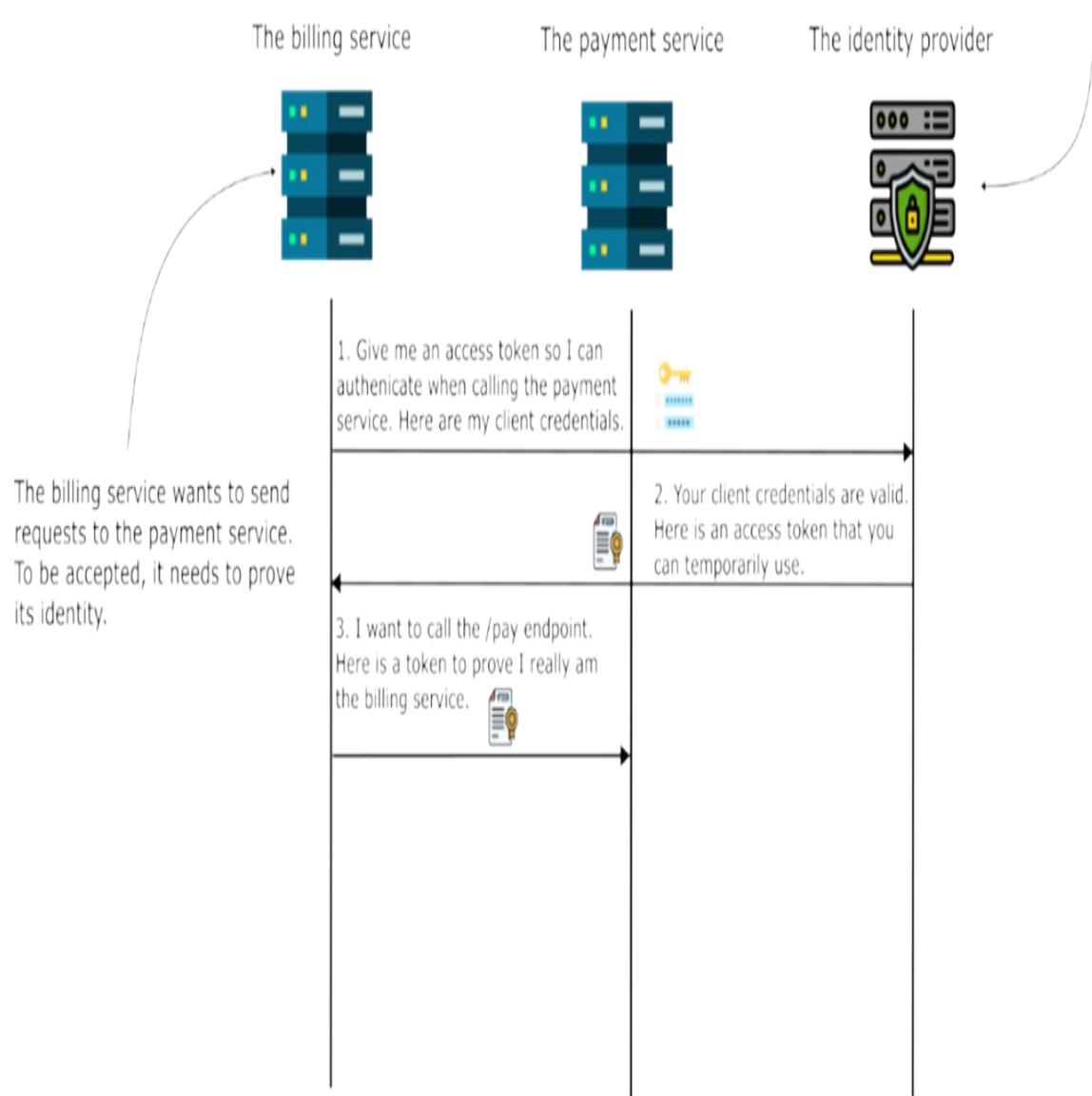
16.2.1 Using the OAuth2 client credentials grant type

One of the most robust approaches is the *OAuth2 client credentials flow*. We have discussed OAuth 2 and OpenID Connect in chapters 12 and 13. In these chapters, we focused on authorization flows that involve a user, such as the authorization code grant type. But OAuth2 also defines userless authorization flows for service authentication. OAuth 2 client credentials is what services would use to authenticate one against the other.

In this model, a service (like order-service) authenticates with an identity provider (IdP) (such as Keycloak or Auth0) and gets a token, usually a JWT. This token is then attached to any outgoing requests to another service (like billing-service). The receiving service validates the token and reads its identity information (figure 16.5).

Figure 16.5 The client credentials flow. Each service obtains a token from an identity provider by providing their valid client credentials. They use this token to send requests to other services in the system. The token has a limited lifespan and a service needs to get a new token from time to time.

The identity provider (IdP) is responsible with validating service identities and provides tokens to services. Services use these to prove their identities against other services in the system.



Usually, implementing OAuth 2 flows is straightforward today. Frameworks such as Spring Security provide out-of-the-box configurations to allow you to develop this with minimal effort. Frameworks also give you excellent flexibility for customization. Listing 16.1 shows you an example of the Spring Security configuration for a service that uses the OAuth2 client

credentials grant type.

Listing 16.1 Using Spring Security to configure the client credentials grant type

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-client:
            client-id: your-client-id #A
            client-secret: your-client-secret #B
            authorization-grant-type: client_credentials #C
            scope: read,write #D
        provider:
          my-provider:
            token-uri: https://auth-server.com/oauth2/token #E
```

In this book we focus on the best practices rather than deeply going into using frameworks such as Spring Security. But if your intention is to learn deeper Spring Security properly, I recommend you once more the Spring Security in Action, second edition by Laurentiu Spilca (Manning, 2023).

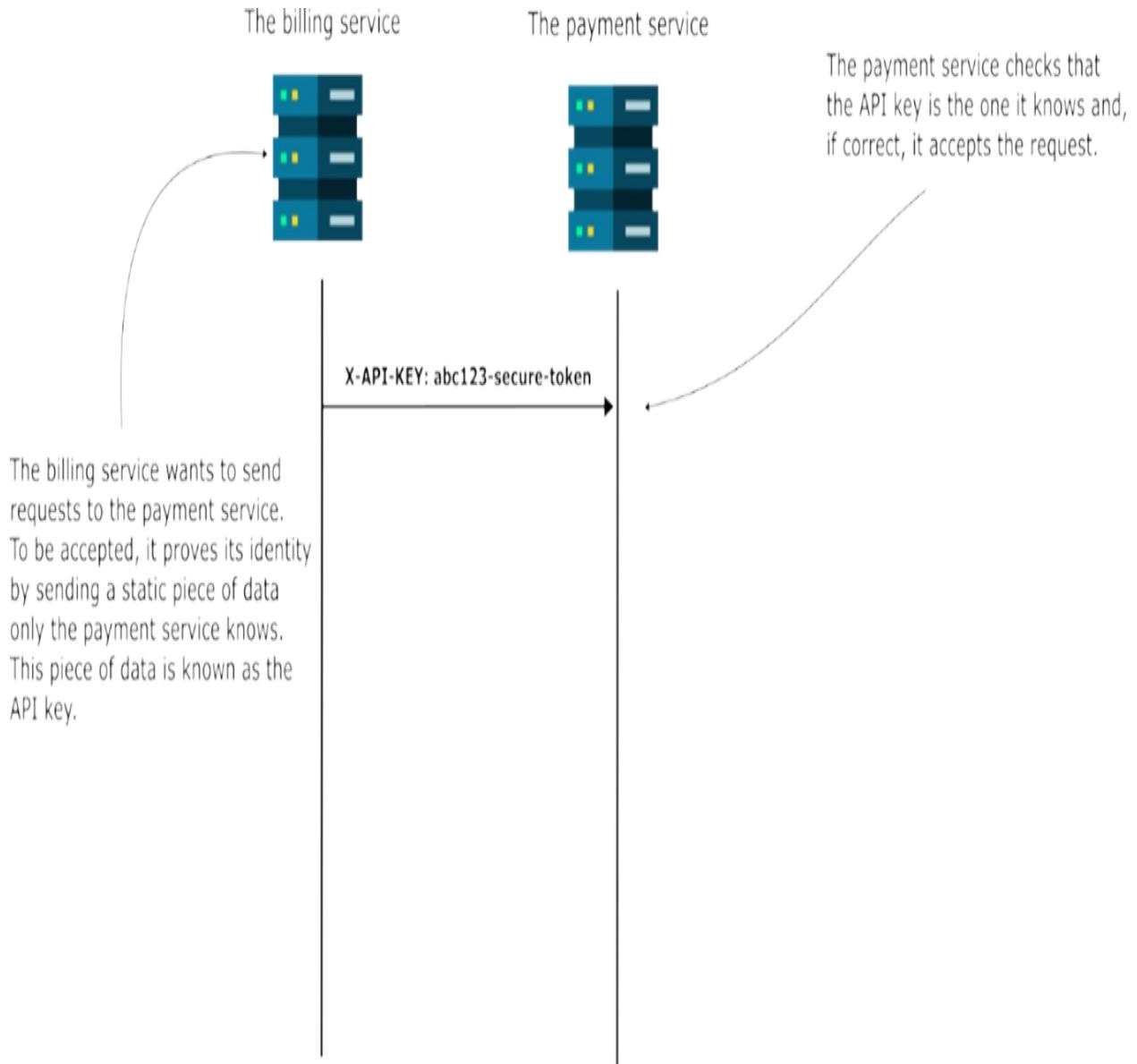
16.2.2 Using API Keys

API keys are one of the simplest ways to identify and authorize a service. An API key is just a unique string, like a password, that one service includes in its request to another service. If the receiving service recognizes the key, it accepts the request. If not, it rejects it.

This approach is very easy to implement. In fact, many internal systems or hobby projects start with API keys because you don't need a complex setup; just generate a key, put it in your app's configuration, and check for it in incoming requests (figure 16.6). For example, order service could send a request to billing service with a custom header like:

X-API-KEY: abc123-secure-token

Figure 16.6 The API key authentication approach is very simplistic. Both sides know a secret they use to prove who they are when sending the requests.



The billing service wants to send requests to the payment service. To be accepted, it proves its identity by sending a static piece of data only the payment service knows. This piece of data is known as the API key.

The payment service checks that the API key is the one it knows and, if correct, it accepts the request.

Listing 16.2 shows you a simple code example that implements the logic that validates the API key.

Listing 16.2 Validating the API Key

```

@PostMapping("/process")
public ResponseEntity<String> processRequest(
    @RequestHeader("X-API-KEY") String apiKey) {
    if (!"abc123-secure-token".equals(apiKey)) { #A
        return ResponseEntity
            .status(HttpStatus.UNAUTHORIZED)
    }
}

```

```
        .body("Invalid API Key");
    }

    // continue logic

    return ResponseEntity.ok("Invoice processed");
}
```

Note

In listing 16.2, the logic appears in the controller to make it straightforward to understand. Normally, in a real-world implementation, such logic must be extracted into an aspect or using an alternative approach that ensures avoiding code duplication.

Even though they're easy, API keys have important weaknesses. The biggest problem is that they can be easily leaked (e.g. in logs, in GitHub repos, in error messages, or inside container images). If someone gets hold of a valid API key, they can impersonate the service and access everything that key allows.

Another issue is that API keys are static. Unless you manually rotate them (which is often forgotten), they remain valid forever. And because they're just simple strings, they don't carry any information. They don't say who made the request, when, or why. You can't limit them to specific actions unless you write a lot of custom logic.

Also, headers like X-API-KEY can be spoofed if you're not using HTTPS or mTLS. If your internal network gets breached or misconfigured, another service (or attacker) can simply send a fake request with the same header and trick your service into accepting it (remember how our hacker Alexey proceeded at the beginning of this chapter).

API keys can be acceptable in tightly controlled environments, like internal tools that are not exposed to the internet, and where traffic is already protected by encryption (e.g. mTLS). They're also handy for temporary setups, demos, or very simple systems with few moving parts.

But for production systems, especially in microservices or multi-tenant

architectures, API keys should be replaced with more secure options like JWTs, OAuth2, or mutual TLS, where identity is harder to fake and easier to manage securely.

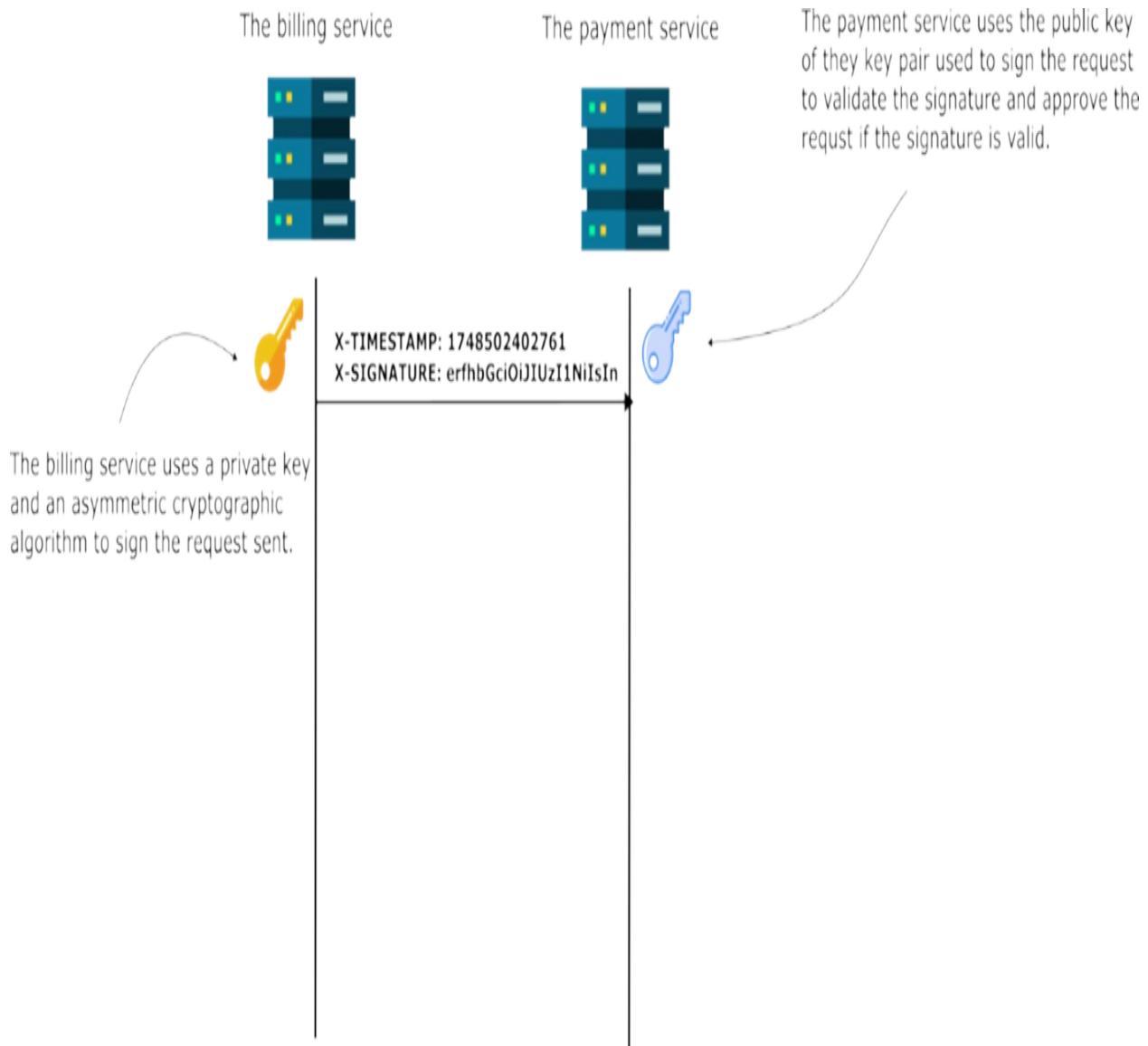
16.2.3 Using custom authorization logic

Sometimes, standard tools like OAuth2 or JWTs don't fit your exact use case, or you need tighter control over how services prove who they are, but we want to avoid less secure approaches such as the API keys. That's where custom authorization logic comes in. To make things more secure, we can use asymmetric keys (see chapter 7 for a refresher).

Here's how it works: the calling service (let's say inventory service) has a private key that only it knows. Before it sends a request to another service (like product service), it takes some important parts of the request, like the path, a timestamp, or even the request body, and creates a digital signature using that private key. That signature is added to the request as a header.

When product service receives the request, it doesn't trust the service name blindly. Instead, it uses the public key (which is safe to share) to verify that the signature is valid. If the signature checks out, the request is accepted (figure 16.7). If not, the request is rejected, because it might have been forged or tampered with.

Figure 16.7 Key pairs and asymmetric cryptographic algorithms can be used to sign the requests made by services to ensure better security. In this approach the service signs the request it makes with a private key. The service receiving the request can validate the signature using the public key and decide whether it approves or rejects the request.



This method adds a good layer of safety: even if someone sees the request, they can't copy or fake it without the private key. Listing 16.3 shows you what would a simplistic implementation for creating such a signature would look like in Java code.

Listing 16.3 Signing a request

```
public class RequestSigner {
    private final PrivateKey privateKey;
    public RequestSigner(PrivateKey privateKey) {
        this.privateKey = privateKey;
```

```

    }

    public String signRequest(String path,
        String timestamp) throws Exception {
        String message = path + "|" + timestamp;

        Signature signature = Signature.getInstance("SHA256withRS");
        signature.initSign(privateKey);
        signature.update(message.getBytes(StandardCharsets.UTF_8)

        byte[] signedBytes = signature.sign();
        return Base64.getEncoder().encodeToString(signedBytes);
    }
}

```

In simple words, to use the `RequestSigner` implementation, a service would augment each request by adding the signature to an HTTP header. The next code snippet shows how that piece of code would look like. Most probably, an app would decouple this (probably into an aspect in the case of a Spring app) such that it doesn't have to repeat for each endpoint implementation.

```

String path = "/adjust-stock";
String timestamp = Instant.now().toString();

String signature = signer.signRequest(path, timestamp);

HttpHeaders headers = new HttpHeaders();
headers.add("X-Timestamp", timestamp);
headers.add("X-Signature", signature);
headers.add("X-Service-Name", "inventory-service");

```

Once a request includes a signature, the receiving service only needs to validate it to confirm that the request came from the expected application and hasn't been tampered with. This step ensures both authenticity and integrity. Listing 16.4 shows a straightforward implementation of a utility that verifies the signature on an incoming request.

Listing 16.4 Validating a request

```

public class RequestVerifier {

    private final PublicKey publicKey;

```

```

public RequestVerifier(PublicKey publicKey) {
    this.publicKey = publicKey;
}

public boolean verifyRequest(String path,
    String timestamp,
    String receivedSignature) throws Exception {
    String message = path + "|" + timestamp;

    Signature verifier = Signature.getInstance("SHA256withRSA"
        verifier.initVerify(publicKey);
        verifier.update(message.getBytes());
        byte[] signatureBytes =
            Base64.getDecoder().decode(receivedSignature);

        return verifier.verify(signatureBytes);
    }
}

```

Remember that keys need to be stored securely and not hardcoded within the app, source code, or container. One of the biggest security mistakes is hardcoding private or public keys directly in your source code or configuration files. If your code is ever pushed to a public repository, or even shared within a company, those keys may be exposed. Instead, store sensitive keys in a secure vault such as HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or even environment-specific secret stores like Kubernetes Secrets (with encryption enabled). These tools protect your keys and give you options like automatic rotation, access logging, and fine-grained permissions. By keeping keys outside the application code, you reduce the risk of accidental leaks and simplify key management over time.

Another thing you need to think about when using this approach is replay attacks. Just verifying a signature isn't enough. If an attacker gets hold of a signed request, they could try to re-send (replay) that same request later, even hours or days after it was originally sent. To prevent this, include a timestamp when signing the request (as you observe in our example), and validate it on the receiving side. A common practice is to allow a time window (for example, +/-5 minutes from the server clock). If the timestamp is too old or too far in the future, reject the request. This ensures that even if someone sees

a signed request, they can't re-use it outside that time window, adding a strong layer of protection.

Also you might want to add more data for authorization. By default, you might sign only the URL path and timestamp, but that leaves some room for tampering. To make your request signature more robust, include additional fields such as the HTTP method (GET, POST), query parameters, and even a hash of the request body. For example, signing the full string "POST|/adjust-stock|timestamp|SHA256(body)" ensures that even tiny changes to the body or method will result in an invalid signature. This helps protect against man-in-the-middle modifications or subtle replay attempts, making your authorization mechanism much harder to bypass.

Custom authorization logic gives you full control and flexibility, but it also places the burden of security on each individual service. As systems grow, this can become harder to manage and easier to get wrong. In the next section, 16.3, we'll focus on infrastructure-level authorization, where security is enforced by the platform itself, using tools like service meshes, API gateways, and workload identity. This approach can simplify enforcement, improve consistency, and reduce the risk of human error.

16.2.4 Exercises

5. How does app-level service authorization work?
6. What's the OAuth2 client credentials flow used for?
7. Why are API keys risky?
8. How does custom request signing work?
9. What's a replay attack and how do you stop it?

16.3 Implementing service authorization at the infra-level

So far in this chapter, we've looked at how services can take responsibility for authorization, such as validating tokens, checking headers, and making decisions in application code. That works well, but as systems grow, it becomes harder to ensure every team follows the same rules, and that every

service is secure in the same way.

That's where infrastructure-level authorization comes in. Instead of burdening each service, we move the responsibility to the platform by using tools like service meshes or API gateways. These components act like security guards at the gates, checking who's calling and what they're allowed to do before the request ever reaches your application logic.

This approach brings centralized policy enforcement, stronger identity guarantees, and less room for human error. In this section, we'll look at how this works in practice, when to use it, and what tools you can rely on to make it both secure and scalable.

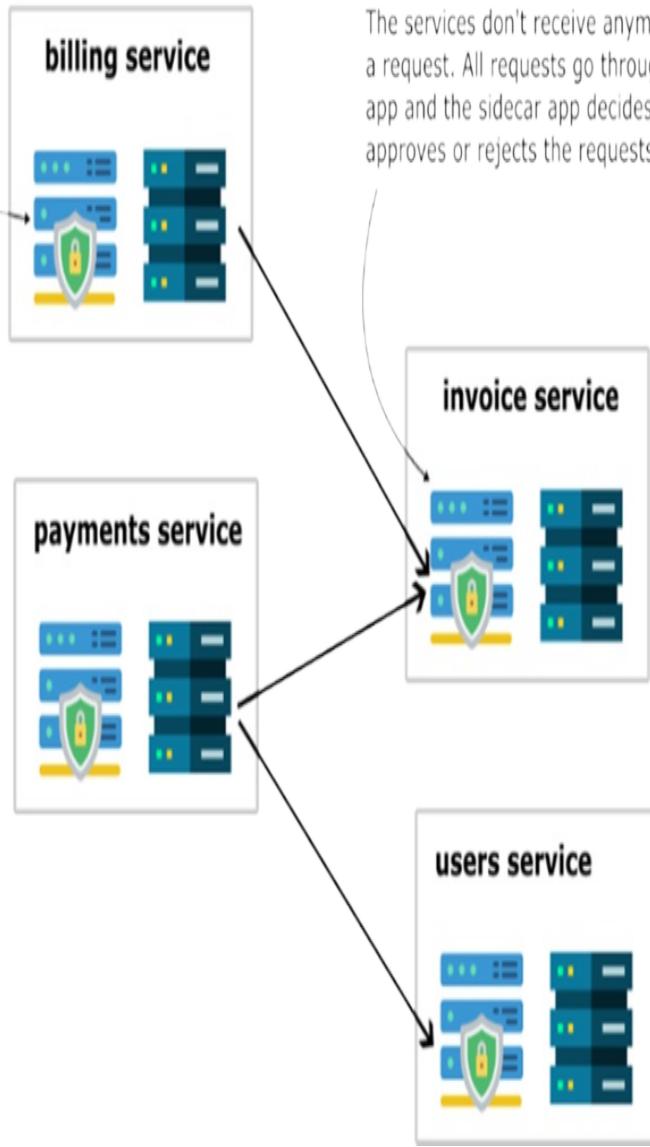
In this section, we'll discuss two approaches most large systems use when applying infrastructure-level service authorization. These approaches can be used together. As this section discusses, you can combine them to give your system excellent strength when needed. We'll talk about:

- *Service Mesh with mTLS (e.g., Istio, Linkerd)* - A service mesh transparently secures service-to-service communication by encrypting traffic and verifying service identity using mutual TLS.
- *API Gateways with Identity Enforcement (e.g., Kong, Apigee)* - API gateways act as entry points that authenticate requests, validate tokens, and enforce access control before traffic reaches backend services.

A *service mesh* is an infrastructure layer designed to manage, observe, and secure communication between services in a distributed system. It takes over responsibilities that traditionally lived in application code like service discovery, traffic routing, retries, timeouts, encryption, and access control, and moves them into a dedicated network layer. This is typically achieved using lightweight sidecar proxies deployed alongside each service, which intercept and manage all incoming and outgoing traffic (figure 16.8).

Figure 16.8 In a service mesh, each service is deployed together with a sidecar app. The sidecar app is the one that validates the requests which go towards the app it protects.

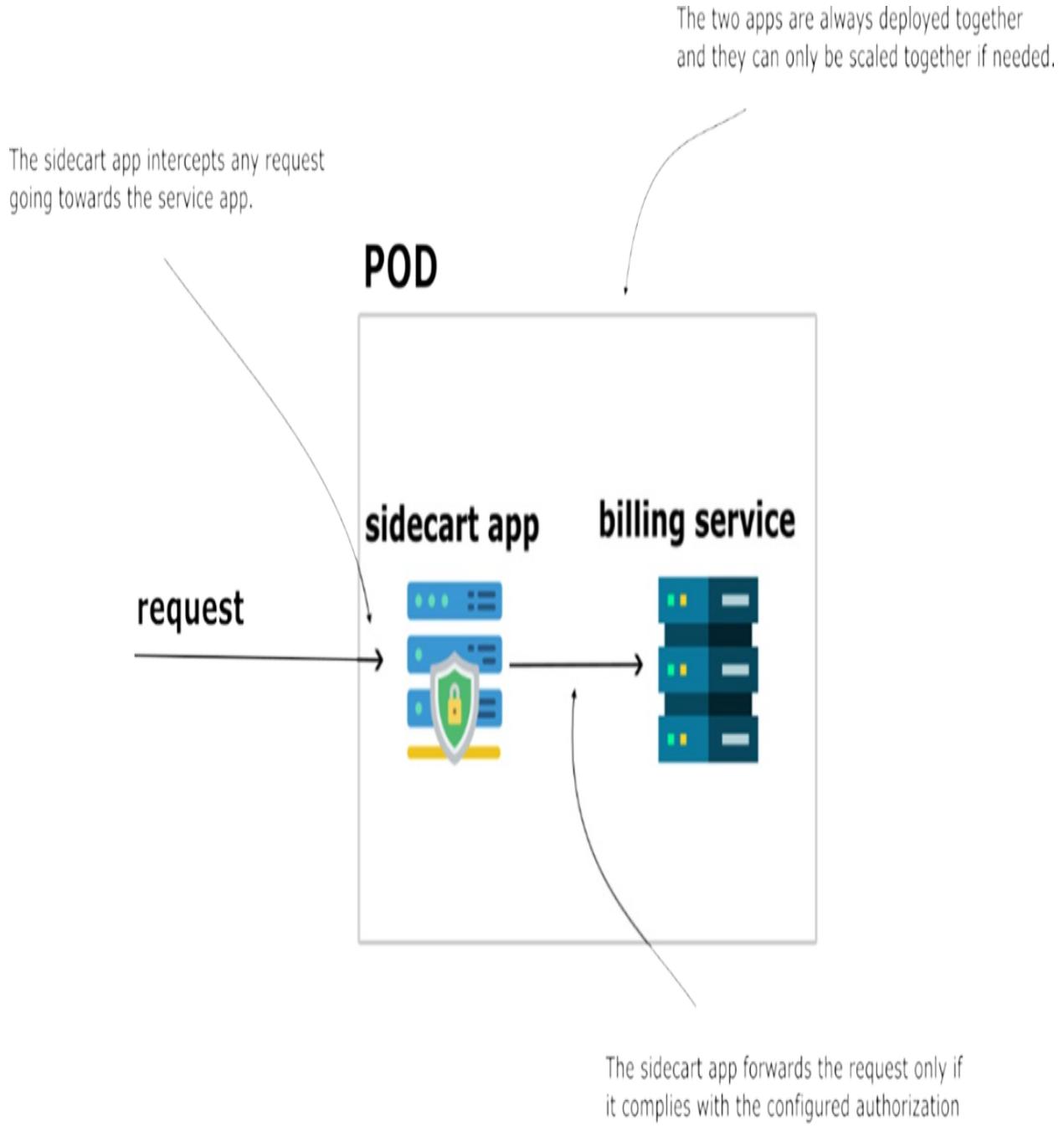
In a service mesh, each service is deployed together with a small app, which we name a sidecar app, which takes the responsibility for the requests authentication.



To achieve the needed isolation both our service and the sidecar app are deployed in separate containers as part of the same pod.

Imagine every service in your system has a tiny security guard (called a sidecar proxy) standing next to it (figure 16.9). When a service wants to talk to another service, it doesn't talk directly. Instead, it tells its guard, “Please send this message.” The receiving service has its own guard too, who checks, “Who are you? Do you have the right badge? Is your message signed?” Only then does the message go through.

Figure 16.9 The anatomy of a pod. The two apps, the sidecart app which takes care of the authorization and the service are always deployed together and can only be scaled together. The sidecart app intercepts all the requests and decides if it forwards the request to the service based on the configured authorization rules.



These guards handle things like:

- Encrypting traffic between services (using mTLS)
- Verifying service identity (using certificates)
- Applying access rules (like “only service A can talk to service B”)

Service meshes like Istio, Linkerd, or Consul Connect make this happen automatically. You don't need to write the security logic into your app. The mesh does it for you, consistently across all services.

When you use infrastructure-level authorization, like with a service mesh, security decisions happen outside the application code. The mesh sits between services, handling things like encryption, service identity, and access rules. One big advantage here is that you don't need to write or maintain this logic in your own code. It all happens automatically! The mesh applies the same rules across all services, which means fewer mistakes and more consistent behavior. It also encrypts all traffic between services and checks that each one is really who it claims to be, usually using mutual TLS (mTLS). This makes it easier for security teams to manage policies from one place without needing to involve every developer.

However, this approach isn't perfect. Setting up a service mesh can be complex. It adds extra pieces to your infrastructure and requires learning new tools and concepts. You also need more computing resources because every service runs with a sidecar proxy. And while the mesh is good at enforcing general rules (like “only service A can talk to service B”), it doesn't know anything about your business logic. For example, it can't decide if a user should be allowed to cancel an order or approve a refund. Such logic still needs to live inside your application.

On the other hand, with application-level authorization, your service handles the decision-making itself. This means you can make very detailed choices based on the content of the request, the user's role, or other internal rules. It gives you flexibility, and you can build exactly the kind of access control your application needs. Developers are in full control, and changes can be made quickly, right in the code.

But this flexibility comes with a cost. Every service has to implement its own authorization logic, and if even one service forgets to check something, it can create a big security hole. It's also harder to track what rules are in place

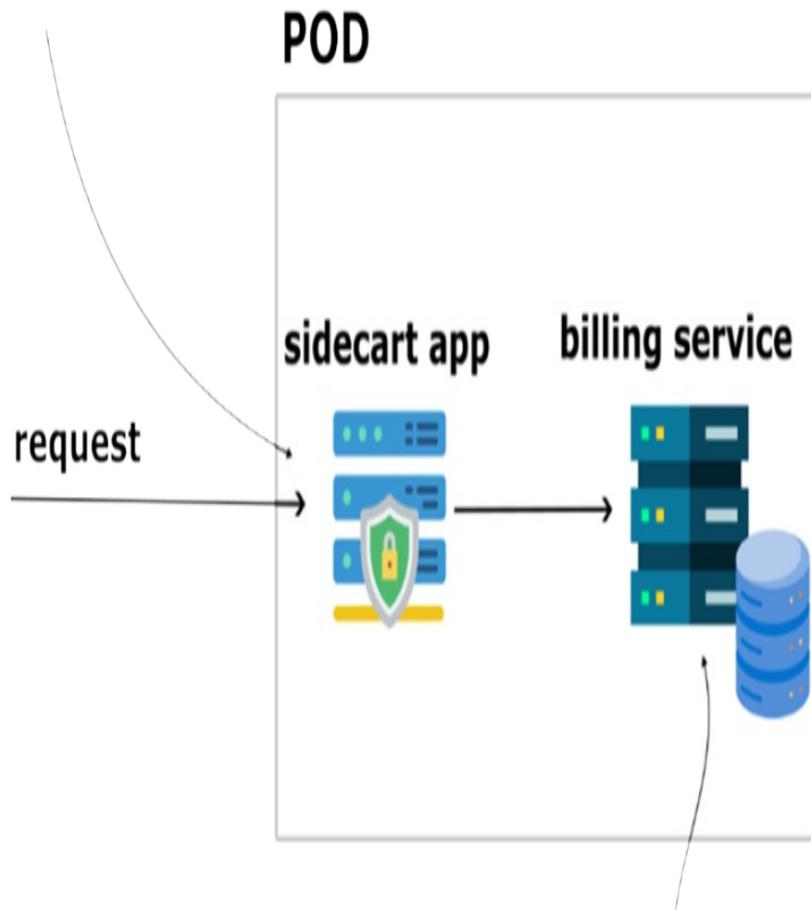
because the logic is spread across different services and codebases. Often, teams write the same code repeatedly, like checking tokens or validating headers, which adds to the maintenance burden.

In short, infrastructure-level authorization is great for consistency, simplicity, and strong baseline security, especially in large systems. Application-level authorization is better when you need detailed, business-aware decisions that depend on the content or purpose of the request. In practice, many teams use both together, letting the infrastructure enforce basic identity and access, while the app adds finer rules on top.

If you want to build a secure and reliable system, using both infrastructure-level and application-level authorization together is one of the smartest choices you can make (figure 16.10). These two layers solve different problems, and when combined, they give you much stronger protection than either one alone.

Figure 16.10 Infrastructure authorization is often combined with app-level authorization. In certain cases, some authorization rules are more business context-specific and need to be defined in the service with access to the needed data.

The sidecart app intercepts any request going towards the service app.



However, the app can still apply certain authorization rules. So the app may reject the request even if the sidecart app approved it. This is because the service usually has more context than the sidecart app (like access to a database), and may apply different rules that are more business-specific.

The infrastructure layer, such as a service mesh or API gateway, is great for handling the basics. It ensures that only trusted services can talk to each other, that the traffic is encrypted, and that every request comes from a known source. This helps stop attackers or unauthorized services before they even reach your code. It's like having a security guard at the door who checks IDs before letting anyone into the building.

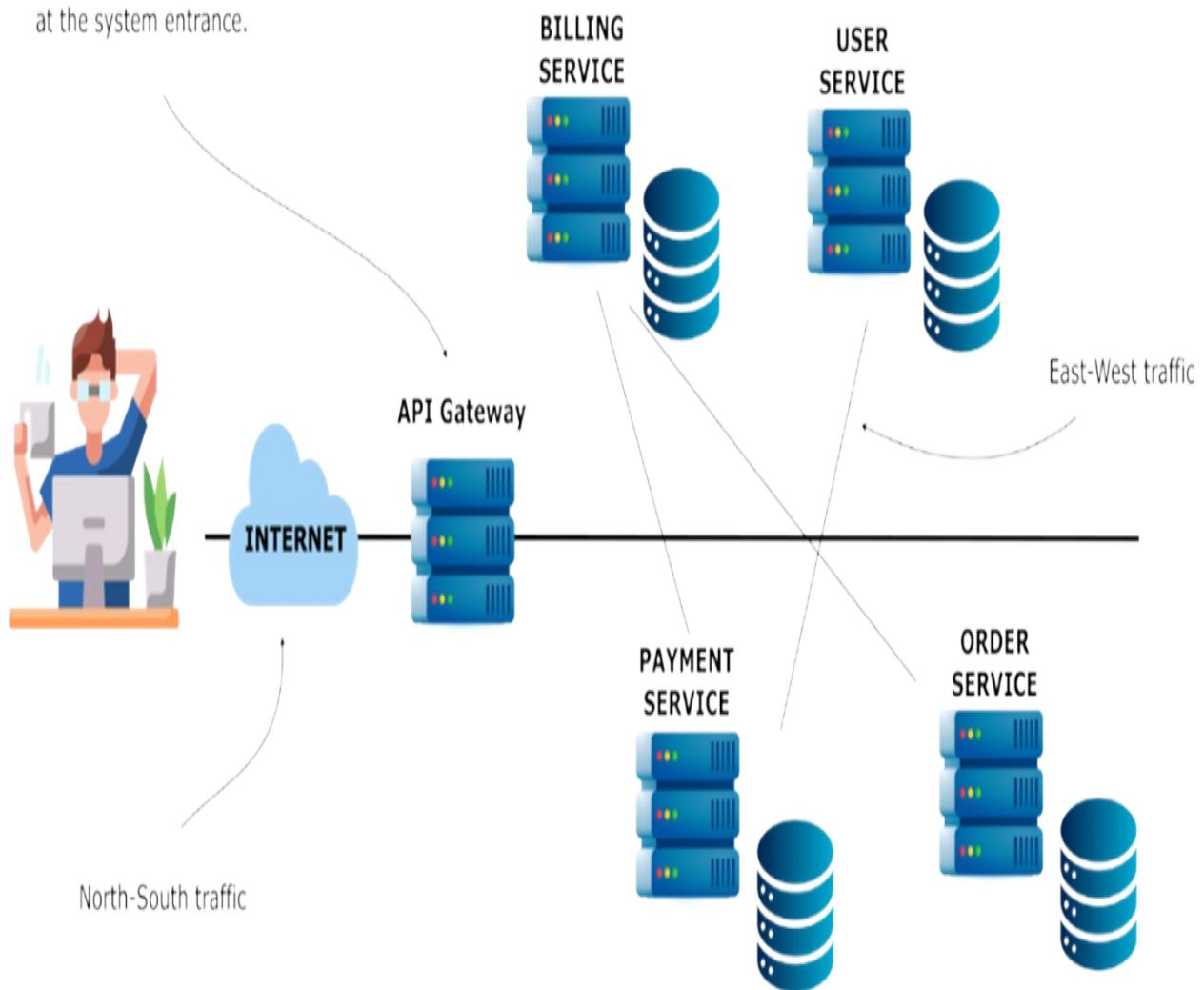
But once a request gets past that door, you still need to check what that request is actually trying to do. That's where the application layer comes in.

Your application can look at the request in more detail like checking the user's role, validating tokens, or applying business-specific rules. For example, it can decide whether a user is allowed to cancel an order, or whether a certain service has permission to update data. This is something the infrastructure layer simply can't see or understand.

Another highly encountered approach is applying authorization at the API Gateway (figure 16.11). An API gateway is like the front door to your system. It's the first thing a request hits before reaching any of your internal services. Think of it as a smart doorman to your system: It checks who's coming in, what they're allowed to do, and where they should be sent.

Figure 16.11 An API Gateway is designed to guard the system entrance. Besides security, it sometimes also takes other responsibilities, such as observability (tracking the requests that go in and out) and service discovery (finds out to which service the request is addressed).

An API Gateway applies authorization rules at the system entrance.



When we talk about identity enforcement at the API gateway, we mean that the gateway is responsible for authenticating and authorizing requests before they go any further. This means:

- Checking who made the request (identity)
- Validating tokens (like JWTs or OAuth2 access tokens)
- Applying policies (like rate limits, scopes, or IP restrictions)

For example, a gateway like Kong, Apigee, or AWS API Gateway can be set up only to allow requests that include a valid token from a trusted identity provider. If the token is missing or invalid, the request is blocked immediately, before it reaches your services.

This is powerful because it centralizes security in one place. You don't need to add token validation logic to every microservice. The gateway handles it for you, consistently and efficiently. It also keeps your services cleaner and more focused on business logic, not authentication code.

API gateways also help with North-South traffic. This is how we name requests coming from outside your system (like browsers or mobile apps). They are a secure barrier between the public world and your internal architecture. You can also combine them with service meshes to protect East-West traffic (between internal services).

Definition

North-South traffic represents communication coming from external systems, while East-West traffic is a way to commonly refer to internal communication between services.

Comparing the API Gateway to the service meshes approach, an API gateway is the first thing a request hits when it enters your system from the outside. Like requests coming from a browser, a mobile app, or a third-party client. It sits at the edge of your system and acts like a gatekeeper. It checks who's making the request, whether they're allowed in, and where the request should go.

In contrast, a service mesh works inside your system. It manages how your own services talk to each other. For example, it secures the communication between your order-service and billing-service. The mesh adds a small "helper" (called a sidecar proxy) next to each service. These sidecars handle security, routing, and monitoring without changing your app's code. The service mesh makes sure every internal request is secure, encrypted, and coming from a verified service.

So, while both API gateways and service meshes help enforce authorization, they do it in different places and for different purposes. The API gateway protects the system from outside traffic. It validates tokens like JWTs, applies rate limits, and blocks unauthorized access right at the door. The service mesh, on the other hand, focuses on internal trust—making sure that even inside the system, only approved services can talk to each other, and that all

traffic is encrypted using mutual TLS.

In many systems, you'll find both working together. The API gateway handles requests from users or clients and forwards them safely inside. Then, the service mesh takes over to control and protect traffic between services. Used together, they form a strong security model: the gateway keeps intruders out, and the mesh makes sure everyone inside behaves.

16.3.1 Exercises

10. What does a service mesh do for security?
11. How do sidecars in a mesh help?
12. What's the role of an API Gateway?
13. How do service meshes and API gateways work together?
14. What are the pros and cons of moving authorization to the infrastructure?

16.4 Answers to exercises

1. What is service identity and why does it matter?

Service identity is how an app proves who it is to other services.

Without it, any service could pretend to be another, making your system easy to trick.

2. What's the difference between identity and authentication?

Identity says "this is who I am," and authentication checks if that's true.

Both are needed to trust a request.

3. What's a trust anchor?

A trust anchor is someone (or something) everyone trusts to vouch for identities—like a certificate authority or token issuer.

4. Why does Zero Trust apply here?

Because even services inside your system shouldn't be trusted automatically. Every request must prove who it's from.

5. How does app-level service authorization work?

The app checks who is calling and if that service is allowed to do the requested action, usually by verifying tokens or API keys.

6. What's the OAuth2 client credentials flow used for?

It lets one service securely get a token from an identity provider and use

it to authenticate to another service, without a user involved.

7. Why are API keys risky?

They're just static strings. If leaked, anyone can use them. They're easy to spoof, hard to rotate, and carry no useful info.

8. How does custom request signing work?

A service signs its request with a private key, and the receiving service uses the public key to verify the signature. It's safer but takes more work.

9. What's a replay attack and how do you stop it?

It's when someone reuses a valid signed request to trick your system.

You prevent it by including a timestamp and rejecting old or reused requests.

10. What does a service mesh do for security?

It encrypts service-to-service traffic using mutual TLS and checks service identity before forwarding the request, all outside your app code.

11. How do sidecars in a mesh help?

They sit next to each service and handle encryption, authentication, and authorization rules consistently and transparently.

12. What's the role of an API Gateway?

It's the first line of defense for incoming (external) requests. It checks tokens, enforces policies, and blocks unauthorized traffic before it reaches your system.

13. How do service meshes and API gateways work together?

The gateway filters and secures traffic from outside the system (North-South), while the mesh secures traffic inside the system (East-West).

14. What are the pros and cons of moving authorization to the infrastructure?

It gives strong consistency and less human error but can be complex to set up and doesn't handle detailed business rules which remain the app's responsibility.

16.5 Summary

- Service identity is the foundation for securing service-to-service communication.
- At the application level, services validate tokens, API keys, or signed messages to decide if a request should be allowed.

- Application-level authorization gives flexibility and fine-grained control, but it requires every service to implement authorization correctly, which can lead to inconsistencies and security gaps.
- Infrastructure-level approaches like service meshes and API gateways move the responsibility outside of the app, providing consistent enforcement, automatic mTLS, and centralized policy management.
- Service meshes secure internal traffic (East-West), while gateways focus on external requests (North-South).
- Modern systems often combine both approaches: infrastructure handles identity and baseline access, while the app enforces business-specific rules.
- A mixed model (which includes both application and infrastructure level configurations) for implementing service authorization improves security, reduces human error, and supports Zero Trust architectures.
- No single solution fits all systems. Choosing between application and infrastructure-level authorization depends on system complexity, security needs, and team responsibilities.
 - System complexity refers to the number of services, the communication patterns between them, and the diversity of technologies in use. Simple systems might get away with centralized gateways or API keys, while distributed microservices often require more fine-grained, decentralized enforcement at the application level.
 - Security needs vary based on the sensitivity of the data and the threat model. Systems dealing with financial transactions, personal data, or regulated workloads may require multi-layered authorization (e.g., both infra and app level) to ensure defense in depth.
 - Team responsibilities influence where enforcement is best placed. If a platform team manages shared infrastructure but app teams own service logic, it's often more maintainable to delegate coarse-grained rules to the infra layer and keep fine-grained policies within the apps themselves.

17 Taming authorization: RBAC, ABAC, ReBAC

This chapter covers

- Comparing RBAC, ABAC, and ReBAC to get how each model handles access control
- Identifying when a system needs more than just roles to make access decisions
- Designing authorization layers that scale without turning into a maintenance nightmare

“So... you've locked the front door.

Great! But now comes the real question:

who gets to open the fridge?”

Welcome to the wild world of authorization, the part of security that decides not just who you are, but what you're allowed to do. Most systems today aren't just one app or one database. They're a sprawling jungle of services, APIs, functions, dashboards, admin panels, and probably three forgotten Lambda functions you deployed last year and now can't find. And each of these needs to decide:

“Should I allow this request?”

“Can this user see this document?”

“Should I trust this service call?”

Authorization models are how we answer those questions. In this chapter, we'll walk through authorization models with real-world examples, explain where they shine (and break), and look at how authorization plays out in

monoliths, microservices, and clouds. You'll learn how companies actually enforce "who can do what," and what tools and patterns help keep that logic sane.

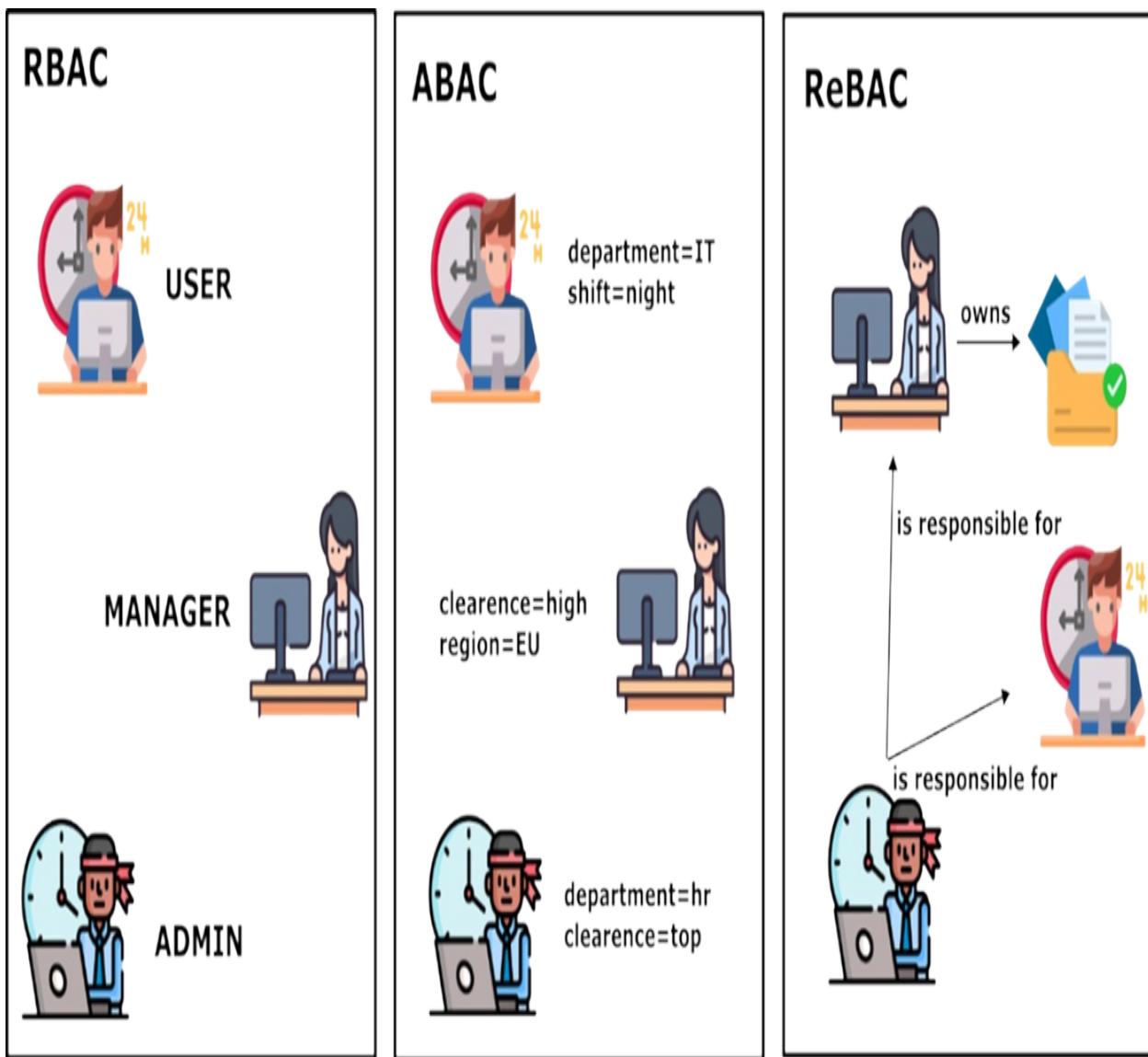
17.1 What are core authorization models

Before we start throwing around tokens, graphs, and policy engines, let's take a step back and ask a simple question: *How do systems actually decide who's allowed to do what?*

In this section, we'll look at the three main strategies used to answer that question:

- *Role-Based Access Control (RBAC)*: "If you have the right badge, you get in."
- *Attribute-Based Access Control (ABAC)*: "You get in if your badge, department, and clearance all match the rules."
- *Relationship-Based Access Control (ReBAC)*: "You get in if you know the right people."

Figure 17.1 Comparing RBAC, ABAC, and ReBAC. RBAC grants access based on predefined roles like user, manager, or admin. ABAC adds flexibility by using user and resource attributes such as department, shift, and clearance level to make decisions. ReBAC focuses on the relationships between entities, for example, who owns a resource or who is responsible for whom, making it ideal for collaborative and delegated access scenarios.



Each model solves the same problem: *controlling access*. But each one does that in its own way. Understanding these models will help you choose the right tool for your system's complexity, growth stage, and security needs.

17.1.1 Role-Based Access Control (RBAC)

RBAC is the simplest, and by far the most common authorization model in use today. It works like this: every user is assigned one or more roles, and each role comes with a set of permissions. When someone tries to access a resource, the system checks: “*Does this user have a role that allows this action?*” If yes, they’re in. If not, denied.

Definition

A *role* is a named collection of permissions that represents a set of responsibilities or access rights within a system. Instead of assigning individual permissions to each user, you assign them a role, like “admin” or “editor”. This role comes bundled with the actions they’re allowed to perform.

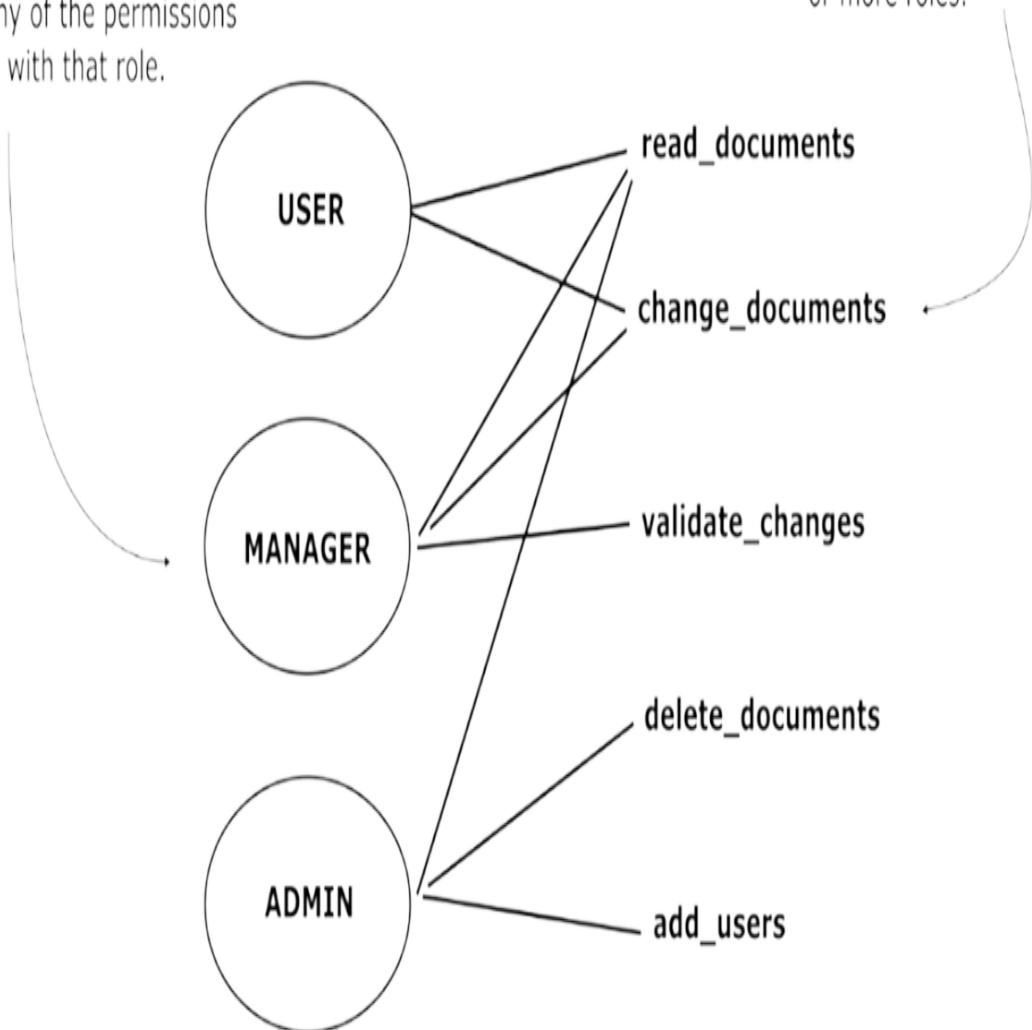
Definition

A *permission* is a specific action or operation that a user is allowed to perform (e.g. “read report,” “edit profile,” or “delete invoice”). Permissions are the building blocks of access control, and roles are typically made up of multiple permissions grouped together.

Figure 17.2 Roles and their associated permissions in RBAC. In role-based access control, each role represents a bundle of permissions. A user assigned to a role automatically inherits all the permissions granted by that role. This structure simplifies access control but can become difficult to manage as permission needs grow more specific.

Roles give you a group of permissions. Someone having a certain role assigned can execute any of the permissions that come with that role.

Permissions are actions that someone can execute. Each permission is granted by one or more roles.



Roles act like security badges, but for apps:

- If you're wearing an “admin” badge, you can go everywhere.
- If you're wearing a “standard” badge, you can look, but not touch.
- If you're “intern”, you can go to the kitchen. Maybe.

Let's say we implement our HR system for Acme Inc. The following might be roles employees can have and based on which they may be allowed (or rejected) certain actions.

- employee: can view their own profile
- manager: can view and edit reports for their team
- admin: can access everything

In a token, the access rule might appear as claim in a token. Next snippet shows you how the JSON defining the body of a JWT could look like.

```
{
  "sub": "alice",  #A
  "roles": ["manager"]  #B
}
```

RBAC is popular mainly because it's easy to understand. Most people already have a basic sense of what a role is. In real life, we all have roles: *teacher, manager, team lead, intern*. Each comes with a set of responsibilities. In a system, it works the same way. You assign someone a role, and that role tells the system what they're allowed to do.

Another reason RBAC is so widely used is that it's supported almost everywhere. Whether you're working in a web framework, a cloud platform, or even designing access in a spreadsheet, you'll find built-in support for roles.

It's also fast and simple to implement. At the technical level, checking a role compares words like "admin" or "editor" to a list. No need to build a complex rules engine or learn a new language to get started. Listing 17.1 shows you a simple example of applying an authorization rule. Frameworks such as Spring Security make the authorization implementation straightforward by eliminating all the boilerplate code. In listing 17.1, you can see how a simple annotation on the method tells the framework who can call that given method.

Listing 17.1 Example of applying an authorization rule on roles with Spring Security

```
@RestController
@RequestMapping("/admin")
public class AdminController {

  @PreAuthorize("hasRole('ADMIN')")  #A
  @GetMapping("/dashboard")
```

```
public ResponseEntity<String> getAdminDashboard() {  
    return ResponseEntity.ok("Welcome to the admin dashboard.");  
}  
}
```

All this makes RBAC a great choice when you're starting to secure access to your application. You can make progress quickly and cover a lot of ground just by assigning people to roles that reflect what they're supposed to do.

Of course, like most simple solutions, RBAC struggles as your system and your organization grow. At first, you might have two basic roles: *user* and *admin*. That's easy. Soon, someone asked:

“*Can we have an admin who can manage invoices but not users?*”

So you create a new role: *invoice_admin*. Then another request comes in:

“*What about someone who manages invoices and contracts, but nothing else?*”

Now you're adding *invoice_contract_admin*.

And it doesn't stop there. This problem is called role explosion. You end up with so many roles that no one can track what they all mean. Your system becomes harder to manage, and people start assigning roles just to make things work, without really understanding what access they're giving.

Definition

Role explosion is a problem that occurs when a system accumulates too many narrowly defined roles in an attempt to cover increasingly specific access needs. Instead of reusing a few well-scoped roles, teams keep creating new ones, like *invoice_admin*, *contract_approver*, *regional_editor_eu*, until the list becomes unmanageable, hard to audit, and prone to inconsistencies.

Even worse, RBAC isn't great when access decisions depend on context. You can't easily answer questions like:

- “*Can this manager edit this specific employee's profile?*”

- “*Can this user see data only from their own region?*

For more precise control like that, you need something more flexible. That's where Attribute-Based Access Control (ABAC) comes in. But before we go there, it's worth saying: RBAC can still take you far if your roles are well designed and your team avoids creating a new one whenever someone has a slightly different job.

17.1.2 Attribute-Based Access Control (ABAC)

ABAC takes authorization one step further by using attributes to make decisions. Instead of just asking, “*Does this user have the right role?*”, ABAC asks, “*Do the attributes of the user, the resource, and the environment match the rules?*”

Definition

An *attribute* is a piece of information about a user, resource, or environment. It may be the user's department, a document's owner, or the time of the request. Attributes are used to define and evaluate access rules in systems that support ABAC.

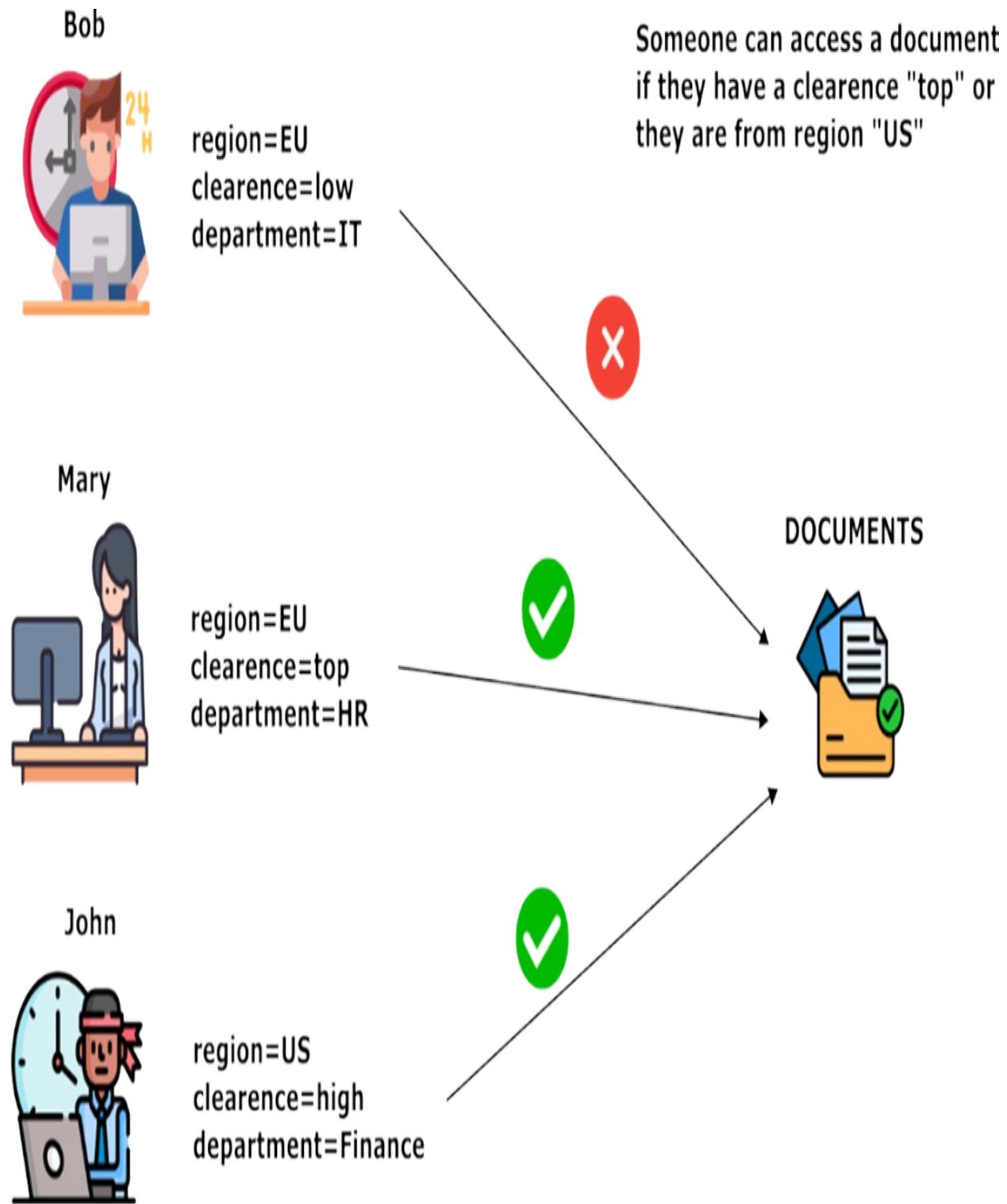
In plain terms, an attribute is just a piece of information. A user might have attributes like `department=finance`, `clearance=high`, or `region=EU`. A resource, like a document or an invoice, might have attributes like `owner_id=1234`, `sensitivity=confidential`, or `created_at=2023-01-01`. And the environment might include attributes like the time of day, the requester's IP address, or whether the request is coming from a mobile device.

With ABAC, you create rules that say things like:

“*A user can access a report only if their department matches the report's department and their clearance level is high enough.*”

Figure 17.3 ABAC rule based on multiple user attributes. In this ABAC example, access to documents is granted if a user has a clearance level of "top" or belongs to the "US" region. Mary meets the clearance requirement, and John meets the region condition, so both are allowed

access. Bob, who has neither "top" clearance nor "US" region, is denied. This highlights how ABAC evaluates access dynamically based on attribute values rather than fixed roles.



This makes ABAC much more flexible than RBAC. Instead of assigning

access through predefined roles, you can describe the exact conditions under which access should be granted.

The biggest benefit of ABAC is its flexibility. You’re no longer limited to predefined roles. You can create fine-grained rules that change depending on who the user is, what they’re trying to access, and the situation around the request.

It’s especially powerful in environments where access depends on real-time context or sensitive data boundaries. For example:

- “Users in the finance department can access invoices marked with `region=EU`.”
- “Operators with `clearanceLevel=HIGH` can download encrypted backups.”
- “Access is denied if the request is made outside of working hours.”

ABAC is also helpful for enforcing least privilege principle, a security principle that says users should have the minimum access they need, no more. With ABAC, it’s easier to grant access only when the attributes truly match, rather than handing out broad roles “just in case.”

- However, the power of ABAC comes with a price: *complexity*.

Writing attribute-based rules can get confusing, especially when dealing with many user types, resource types, and business scenarios. You may start with a few clear rules, but over time the policy set can grow into a maze of conditions that no one wants to touch.

Also, attributes must be accurate and available. If your system relies on `user.region`, but that value is missing or inconsistent; your policies may behave unpredictably or block access when they shouldn’t.

Note

Attributes are only as useful as they are accurate and available. If your system relies on an attribute like `user.region`, but the value is missing, outdated, or inconsistent, access decisions can go wrong.

And because ABAC decisions are often based on live data, performance can become a concern. Evaluating a rule that compares ten different attributes might be slower than a simple role check. Still, ABAC is often the only reasonable choice for systems that need rich, context-aware decisions, especially in regulated industries or multi-tenant environments.

17.1.3 Relationship-Based Access Control (ReBAC)

ReBAC makes access decisions based on relationships between entities such as people, resources, groups, projects, and more. Instead of checking a user's role or matching a set of attributes, ReBAC asks: "*What is the relationship between this user and the thing they're trying to access?*"

Definition

An *entity* is any object in the system that can participate in access control. This includes users, groups, documents, projects, teams, or any resource you want to protect.

Definition

A *relationship* is a connection between two entities that describes how they are linked. For example "Alice is Project Z owns a member of Team X" or "Document Y." These relationships are used to decide whether access should be allowed.

This model works especially well when access depends on who owns what, who is part of what, or who has been granted access by someone else. Imagine a document-sharing app, like Google Docs. A user shouldn't access a document just because they're a "viewer" in general. They should only see it if:

- They own the document
- It's been shared with them
- They're part of a team that the document belongs to

In ReBAC, those relationships like ownership, membership, and sharing are

modeled explicitly. So a policy might say:

“Allow access if the user has a viewer relationship to the document.”

“Doctors can view medical records only for patients in their care.”

“Employees can approve transactions only if they aren’t the ones who initiated them.”

ReBAC shines when access isn’t determined by job titles or static rules, but by how people and resources are connected. That’s especially true in systems with:

- Collaborative features (e.g., shared files, boards, tasks)
- Multi-tenant platforms (e.g., SaaS apps where each customer sees only their own data)
- Organizational hierarchies (e.g., managers can access reports for their direct reports)

This model lets you support powerful use cases like delegation (“Alice can approve on Bob’s behalf”), project-based access (“everyone on Project X can see these dashboards”), or dynamic sharing (“only people I explicitly invited can see this”).

Many modern systems already have these relationships in their databases. They’re just not always using them for access control. ReBAC turns those connections into part of the security model.

But ReBAC also has challenges. First, it requires you to build and maintain a graph of relationships. In many systems you don’t have this by default. You need to know who is connected to what, and how those connections are structured and stored. That could mean extra complexity in your data model and in your access control logic.

Second, performance can be a concern. Answering the question “Is user A related to resource B through some path?” often means querying a relationship graph or policy engine. This can be slower than a simple role or attribute check, especially as your system grows.

Third, reasoning about access can be harder. It's not always obvious why someone has access, especially when relationships are indirect or inherited (e.g., "Alice is in a team that is part of a group that owns the folder").

Still, for systems that need fine-grained, context-aware access, and especially those that allow users to share, collaborate, or delegate, ReBAC is often the best fit. Table 17.1 summarizes the essential characteristics of these models, which we have discussed throughout this section.

Table 17.1 A summary of the three models

	RBAC	ABAC	ReBAC
What access depends on	User's role	Attributes of user, resource, and environment	Relationships between entities
Typical use case	Small apps, admin panels	Regulated systems, dynamic conditions	Collaboration tools, multi-tenant systems
Risk as system grows	Role explosion	Attribute sprawl / policy complexity	Relationship graph complexity
Performance	Fast (simple role checks)	Depends on attribute access and evaluation	May require graph traversal or relationship lookups

17.1.4 Exercises

1. What's the main difference between RBAC, ABAC, and ReBAC?
2. Why does RBAC struggle in large systems?
3. When would you choose ABAC over RBAC?

17.2 Authorization in real-world architectures

At Acme Inc., things started simple. The app was a monolith, and the

backend team added a quick check: only users with the admin role could access the settings page. Easy.

Then came the mobile app. And the reporting service. And the move to microservices. Soon, authorization wasn't just happening in one place. It had to be enforced across five services, each written by a different team. Some checked roles from the JWT. Others pulled user info from the database. One service just... trusted what the previous service said.

One day, a user without the right permissions accessed confidential audit logs through a chain of three services, none of which rechecked the user's access. Nobody noticed until a customer filed a complaint.

After that, Acme's teams sat down to rework their authorization model. They had to decide where checks should live, how to share identity information across services, and how to log who accessed what. It wasn't glamorous, but it turned out to be essential.

So far in this chapter, we've talked about what authorization is and the different models you can use to decide who gets access to what. But in the real world, things aren't so neat. It's one thing to define access rules. It's another to figure out where those rules should live and how they're actually enforced across a system that spans apps, APIs, cloud functions, and maybe a few mystery services running under someone's desk.

17.2.1 Authorization within monoliths

In a monolithic application, all the business logic, data access, and user interactions typically live in the same codebase, and often in the same process. That makes authorization relatively easy to manage: you usually have access to all the data and user context you need, and you can enforce rules right where the action happens.

Monoliths support all three models: RBAC, ABAC, and ReBAC. Usually within monoliths, it's easier to implement than distributed systems because everything is already in one place.

- RBAC is the most common starting point. Most frameworks (like Spring

Security, Django, or ASP.NET) support simple role checks out of the box. For example, it's easy to say if `user.hasRole("ADMIN")` or use an annotation like `@PreAuthorize("hasRole('MANAGER')")`.

- ABAC becomes useful when roles aren't enough. For instance, when you want to restrict access based on the user's department, region, or project. Since the monolith has access to both user and resource data, evaluating attributes is straightforward. You might load both the user and the invoice, then check if `user.department == invoice.department`.
- ReBAC can also be implemented inside a monolith, especially if your domain already includes hierarchical relationships (users, teams, projects). Since all your data is local, you can load the necessary relationships directly from the database and check things like: *"Is this user related to this document through a chain of ownership or membership?"*

Definition

A non-modular monolith is a single codebase where all application parts are tightly coupled, often sharing data structures, services, and logic without clear separation. Changes in one part of the system can easily affect others, and code tends to grow tangled over time.

In *non-modular monoliths*, authorization logic often ends up scattered across controllers, services, or helper classes. This works at small scale but becomes a maintenance headache as the system grows. A developer might add a permission check in one place but forget it in another.

Definition

A modular monolith is still a single application, but it's organized into well-defined, independent modules, each responsible for a specific domain or feature. These modules communicate through clean interfaces, making the system easier to maintain, test, and reason about, even as it grows.

In *modular monoliths*, you can do better. If each module, say, *Users*, *Invoices*, *Reports*, *Teams*, has clearly defined boundaries and owns its own

data and business logic, you can localize authorization rules within the module that knows the context best. For example, the *Invoices* module can contain all the rules about who can create, view, or approve invoices, without needing to rely on other parts of the system. This improves encapsulation, meaning each module becomes responsible for protecting its own domain objects, making the system easier to maintain and audit.

This also avoids the trap of having one giant "security service" that tries to know everything about everything (God Object antipattern). Instead, each module becomes the gatekeeper for its own resources, enforcing access rules based on the data and relationships it manages. It also makes testing easier: you can unit-test authorization rules as part of the module's normal logic without simulating the entire application state.

Best of all, this modular approach helps teams reason more clearly about access control. If someone asks, "Who can edit an invoice?", you know the answer lies in the *Invoices* module, not scattered across controllers, utility classes, or some forgotten *SecurityUtils* file.

Best practice

Avoid sprinkling authorization checks randomly across the code. Instead, centralize them inside clearly named methods or decorators that act as gatekeepers. For example, create a method like `canEditInvoice(user, invoice)` and use it consistently throughout your codebase.

One benefit of doing authorization in a monolith is that you can easily add logging, tracing, or even audit events every time an access decision is made. Since you're not dealing with cross-service communication, you can see and control the full flow of data and decisions.

Tip

Consider logging failed access attempts with enough detail to help troubleshoot without leaking sensitive information. This is often the first sign of a misconfigured role, expired session, or even an attempted attack. For more details on the subject you find an extended discussion on logging in Java apps in *Troubleshooting Java*, second edition by Laurentiu Spilca

(Manning, 2025).

17.2.2 Service-oriented authorization

As systems grow beyond a single codebase and evolve into service-oriented architectures (SOA or microservices), authorization becomes more complex and more important. Now, instead of a single application making decisions in one place, multiple services must decide independently whether a request should be allowed.

In these architectures, services often communicate over the network using REST, gRPC, messaging queues, or event streams. Each service might be developed by a different team, deployed independently, and scaled separately. That means you can't assume that a request reaching a service is safe, just because it passed some check upstream. Every service needs to take responsibility for its own authorization.

- *RBAC* often relies on roles embedded in tokens (e.g., in a JWT). Each service checks roles locally, but the logic can drift over time. One service might treat the manager role as powerful, while another barely uses it. This leads to inconsistent access unless role semantics are well-defined and shared.
- *ABAC* works well when identity tokens carry rich claims (like department, region, or clearance_level). Each service can evaluate these claims against the resources it owns. But again, everyone has to agree on the meaning of these attributes and where to source them from.
- *ReBAC* in service-oriented systems usually means maintaining a central relationship store, like a graph that services can query or defer to when making access decisions. This can be done by calling an authorization service (like OpenFGA or a custom engine), or by embedding relationship data in the request context.

Tip

When going distributed, always assume your service is operating in an untrusted environment. Just because a frontend or upstream service says the user doesn't have access, it doesn't mean you should believe it. Validate

everything.

RBAC in service-oriented systems often relies on roles embedded directly in identity tokens, most commonly in a JWT issued by an authentication server. These tokens might include something like "roles": ["user", "manager"], which each service uses to determine access.

```
{  
  "sub": "1234",  
  "name": "Alice Smith",  
  "roles": ["admin", "manager"], #A  
}
```

At first, this seems straightforward. Each service reads the token, checks if the user has the right role, and proceeds accordingly. But over time, role interpretation tends to drift. One service might consider the manager role as someone who can view and edit team data. Another might treat it as just a read-only role for summary reports. Yet another might not even recognize the manager role at all.

This drift happens because roles are just labels. Without a shared definition, like a documented and agreed-upon meaning for each role, teams start making their own assumptions. This leads to inconsistent access control across services. A user with the same token might be allowed to edit data in one place and denied access in another, simply because each service interprets the role differently.

For example, in a logistics platform, the dispatcher role was meant to allow viewing of shipment routes. But one service also allowed dispatchers to cancel deliveries, because the dev team assumed they needed “full control.” It wasn’t a bug, it was a misunderstanding of the role.

To prevent this:

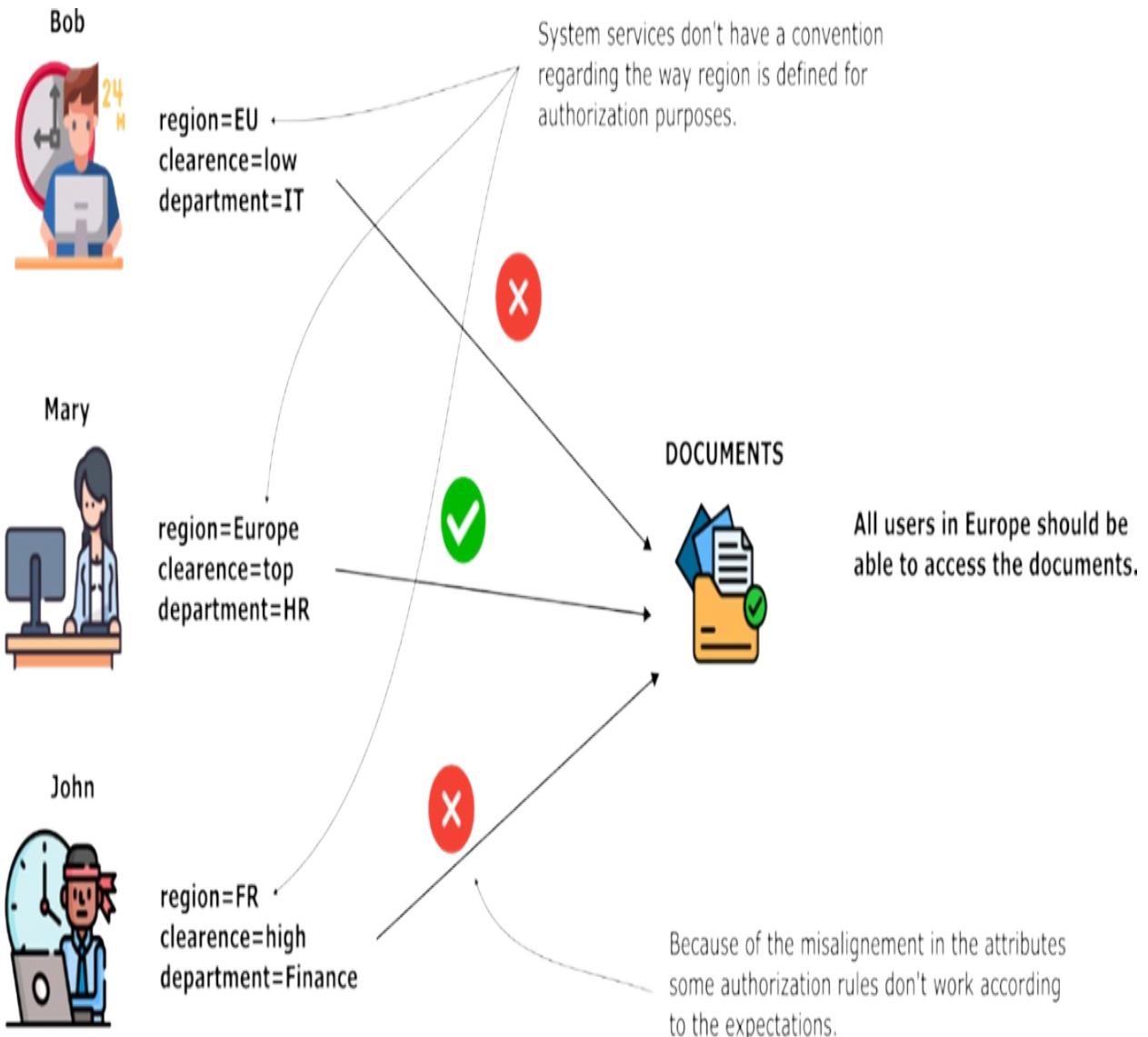
- Define role semantics centrally and document them well.
- Encourage teams to refer to a shared access control spec or contract.
- Avoid using role names alone as the source of truth for permissions. Tie roles to well-defined capabilities or scopes, or combine them with ABAC rules for extra precision.

ABAC works well in service-oriented architectures when identity tokens, like JWTs, carry rich claims, such as `department`, `region`, `project_id`, or `clearance_level`. These attributes give services the context they need to make more fine-grained decisions than just checking roles. For example, a document service can decide to allow access only if `user.department == document.department`, or a reporting API might allow users to fetch data only for the region they belong to.

Each service can evaluate these claims locally, using the attributes directly from the token, without needing to make extra calls to a user directory or central service. This makes ABAC performant and scalable, especially when paired with caching or stateless authorization logic.

But as with RBAC, there's a catch: attribute definitions must be consistent across the system. If one service treats `region` as a country code (e.g., "FR"), another as a continent (e.g., "Europe"), and a third expects it to be a data center name, your access rules will break in unpredictable ways.

Figure 17.4 Attribute inconsistency leads to broken authorization. In this example, all users are meant to access documents if they are from Europe. However, inconsistent attribute values like "EU", "Europe", and "FR" cause authorization failures.



There's also the question of where authorization data actually comes from, and who's responsible for keeping it correct and consistent. This includes things like roles (e.g., admin, manager), attributes (e.g., department=HR, region=EU, clearance=high), or relationships (e.g., "Alice owns document X").

In many systems, this data is expected to come from the identity provider, tools like Keycloak, Auth0, Azure AD, or Okta. These tools often support storing basic roles and user attributes, and can include them in the tokens they issue (such as in a JWT). That works well when you only need a few fixed fields and the user's data doesn't change too often.

However, identity providers don't always have all the context needed for fine-grained authorization decisions. For example:

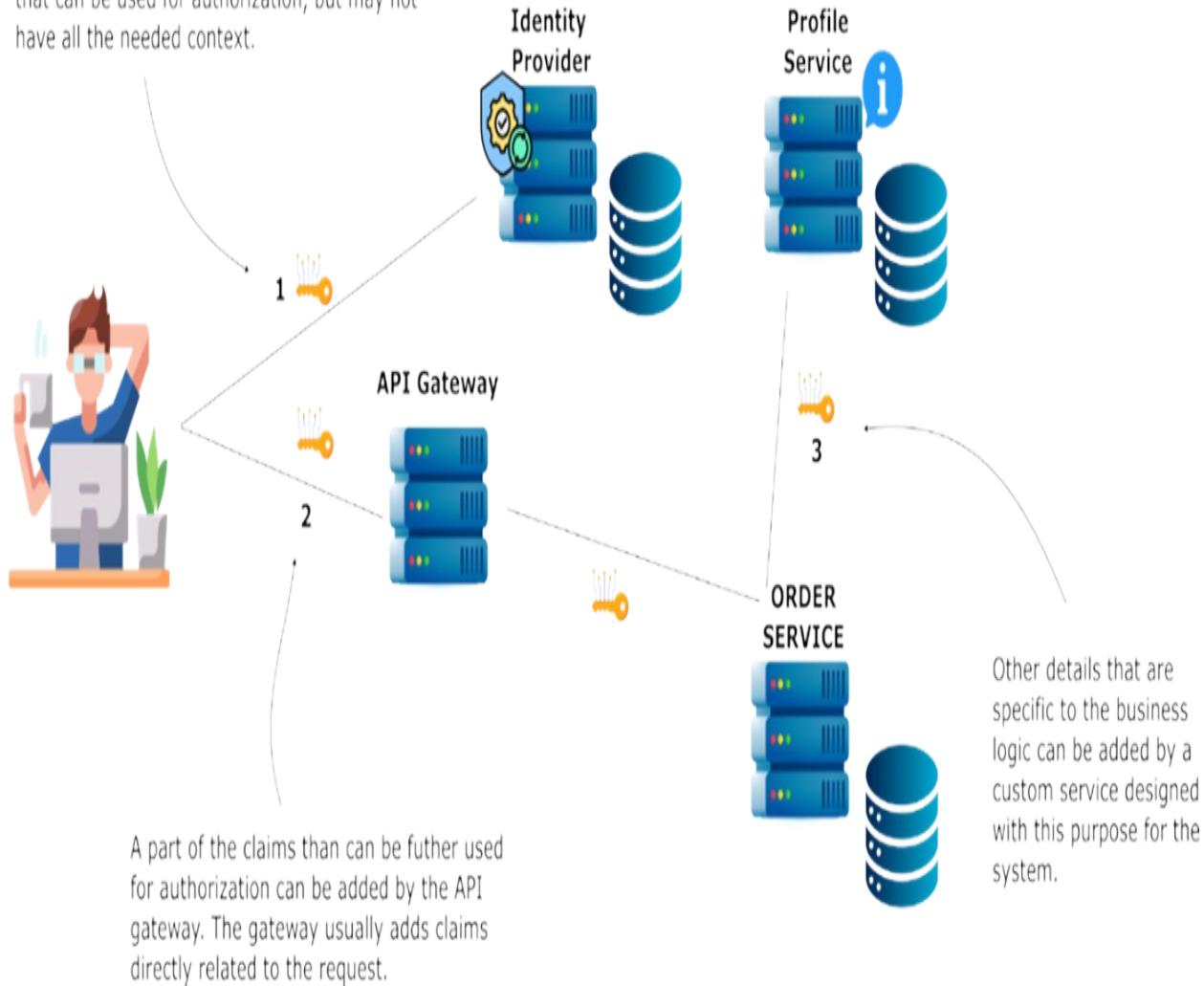
- Keycloak might know that someone is a manager, but it probably doesn't know which projects they're assigned to.
- It may include `department=HR`, but not the fact that they are a reviewer for Document 42.
- It definitely doesn't track things like dynamic relationships between users and resources, like which customers a sales representative currently handles.

In more complex systems, this kind of dynamic or domain-specific data is managed elsewhere. Usually it is managed in a dedicated user profile service, a business domain system, or a centralized authorization service. These systems can be queried in real time, or they can inject data into tokens when needed.

Another common option is to use an API gateway or authentication middleware that enriches the request with additional claims or attributes by calling internal services. This can be useful for attaching context-specific claims (like “`ip_address`”, “`geo_region`”, “`locale`” and so on) just in time, but it also adds complexity and coupling between components. Figure 17.5 shows these system components visually.

Figure 17.5 Enriching authorization claims across system layer. The identity provider issues a token with basic claims, such as roles or user ID. The API Gateway can enhance the token with request-specific claims, like active project IDs or request origin. Finally, backend services (like the Order Service) may fetch additional business-specific data, such as user preferences, access scope, or customer assignments, from a profile service or a custom authorization component. This layered enrichment ensures services receive the right context to make fine-grained authorization decisions.

An identity provider issues the access token.
The identity provider adds some initial claims
that can be used for authorization, but may not
have all the needed context.



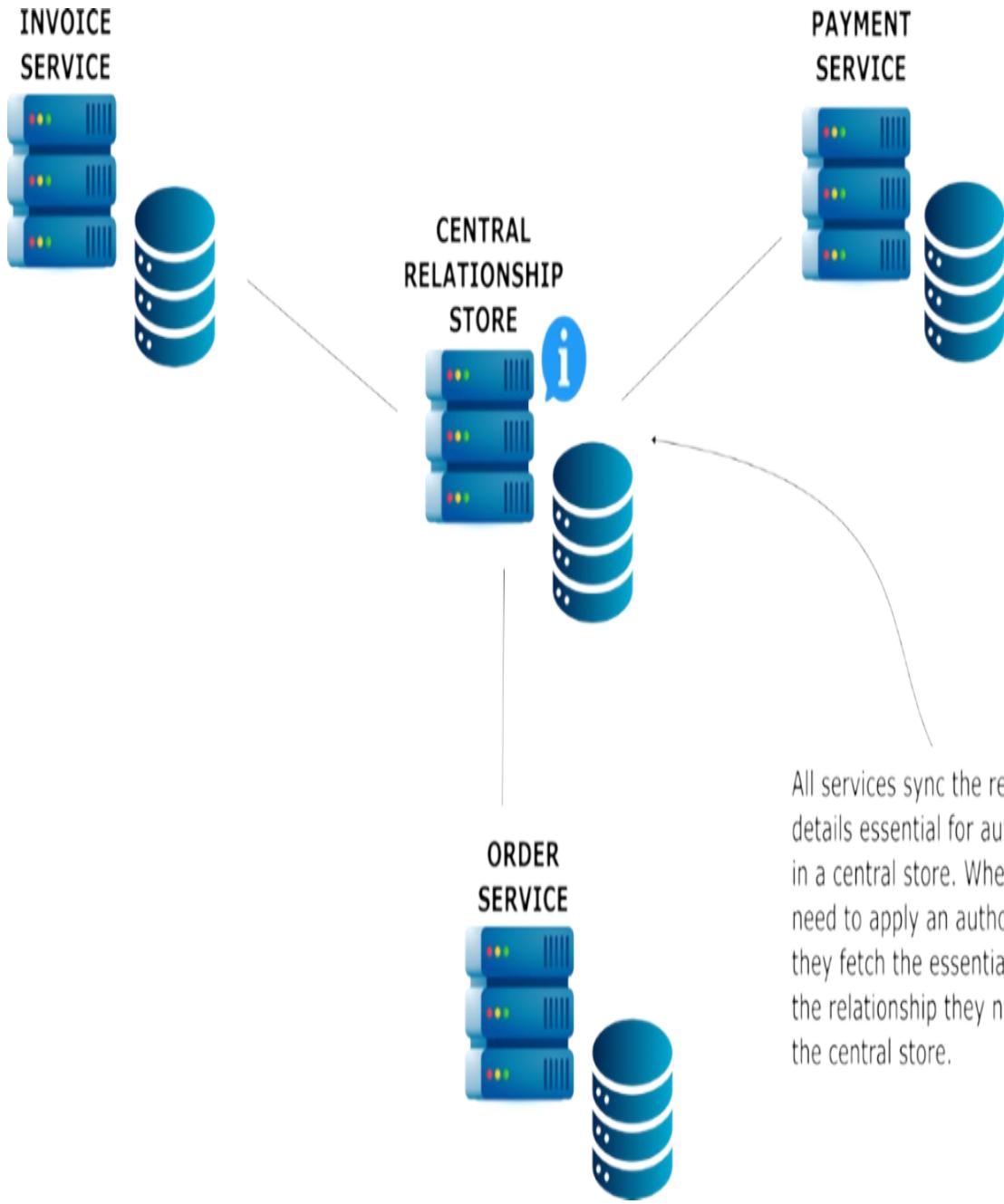
For simple systems, having the identity provider (e.g., Keycloak) store and inject roles and a few core attributes is usually enough. In larger systems, especially those with multi-tenant setups, fine-grained access rules, or fast-changing data, it's better to split responsibilities. Let the identity provider handle authentication and base-level identity and use a dedicated service (like OpenFGA, OPA, or a custom authorization API) to manage fine-grained permissions, attributes, and relationships. This separation keeps your tokens clean, your data sources reliable, and your security logic easier to test, change, and understand.

Tip

To make ABAC work in a distributed system, teams need to agree on a shared attribute vocabulary, a list of well-defined, consistent attributes, their allowed values, and their sources. Treat it like an API contract: access control remains predictable and secure if everyone uses it correctly.

ReBAC in service-oriented systems usually requires a central relationship store. This central store is a system that tracks how users, resources, and groups are connected. Think of it as a graph database where nodes represent entities (users, teams, projects, documents) and edges represent relationships (owns, member_of, shared_with).

Figure 17.6 Using a central relationship store for ReBAC in a microservices architecture. Each service syncs its relationship data (like ownerships, memberships, or associations) to a central relationship store. When a service needs to authorize a request, it queries the store to evaluate whether a valid relationship exists between the user and the resource in question.



All services sync the relationship details essential for authorization in a central store. When any of them need to apply an authorization rule, they fetch the essential details about the relationship they need from the central store.

In a distributed setup, no single service can afford to know the entire relationship graph. So instead, services query this central store when they need to decide whether access should be allowed. For example, a service might ask:

“Is user 123 a collaborator on document 456?”

“Does this user belong to a team that owns this project?”

This is not an easy job to implement, particularly in a microservices architecture. In a microservices architecture, implementing Relationship-Based Access Control (ReBAC) can be tricky because each service owns a piece of the puzzle. For example, the UserService knows who a user is, the TeamService knows who's on which team, and the DocumentService knows who owns each document. But access decisions in ReBAC often rely on relationships that span across these services, such as: "Can Alice view this document because she's part of a team that owns the project the document belongs to?" No single service has all the information to answer that question alone.

To solve this, most real-world ReBAC implementations use a central relationship store. This is a dedicated service, often a graph database or specialized authorization engine that keeps track of how users, resources, and groups are connected. But the relationship store doesn't automatically "know everything."

Each microservice must push its own relationship data to this central store. That means the TeamService needs to publish team memberships, the ProjectService needs to report ownership, and so on. This introduces new complexity: if services forget to sync their data, or if the data becomes stale or inconsistent, access checks may fail or behave incorrectly.

A good idea to simplify such an implementation is using a dedicated authorization service (like OpenFGA [<https://openfga.dev/>], Authzed [<https://authzed.com/>], or a custom ReBAC engine), which evaluates the relationships and returns a yes or no (figure 17.7). The policy itself may be defined using a simple DSL (like OpenFGA's presented in listing 17.2) or a graph-aware language.

Listing 17.2 The OpenFGA DSL

```
model
  schema 1.1

  type user  #A

  type organization  #B
    relations
      define member: [user]  #C
```

```
type document #D
relations
  define owner: [user]
  define org: [organization]
```

Figure 17.7 A service querying the central relationship store to evaluate access. In a ReBAC-enabled architecture, services don't store or compute relationship logic locally. Instead, when the Payment Service needs to verify if user "joe" owns specific invoices, it sends a query to the central relationship store. The store evaluates the relationship graph and returns the answer.

PAYMENT SERVICE



CENTRAL RELATIONSHIP STORE



Does user "joe" own
invoices 123 and 456?

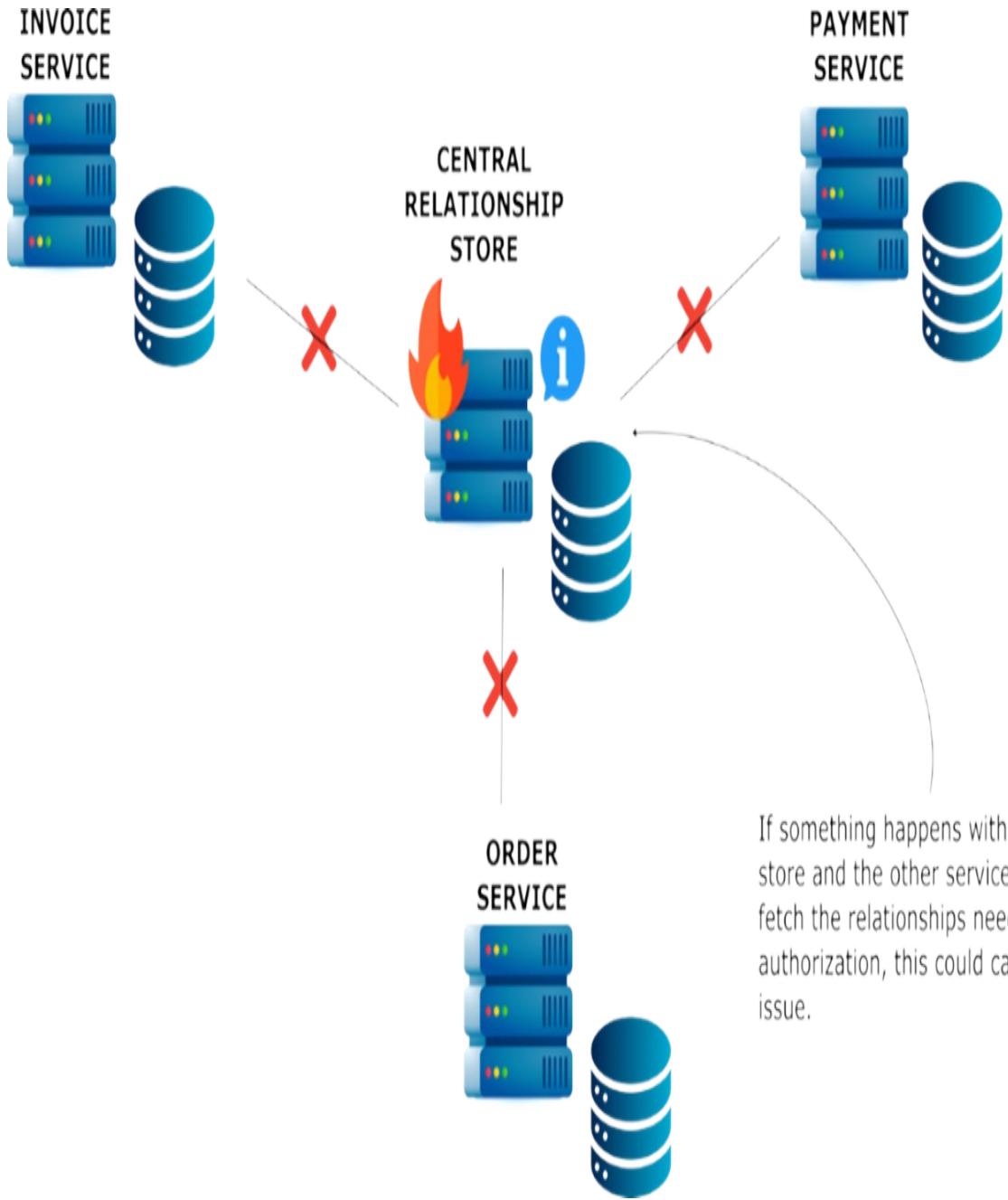
✓ Yes, they do!

Whenever a service needs to validate a certain relationship, it asks the central store.

On the plus side, a centralized store makes authorization consistent, auditable, and easier to reason about. Any service can query the relationship store to ask, “Does user X have a relationship Y to resource Z?”, without needing to reimplement the logic or fetch data from multiple services. This keeps business logic out of microservices and in a centralized place where it’s easier to manage. It also enables complex access patterns like delegation, group-based access, and shared ownership.

However, this approach isn’t free of trade-offs. The biggest downside is that it adds a dependency between your services and the ReBAC store. If it goes down or becomes a bottleneck, access decisions across your system may be delayed or blocked (figure 17.8). You also need to ensure data synchronization is reliable and secure, ideally using events or background jobs to keep the relationship store up to date. Additionally, the ReBAC engine needs to scale independently, especially if many services are making access checks in parallel.

Figure 17.8 Central relationship store as a single point of failure. If the central relationship store becomes unavailable, services relying on it for authorization cannot validate user-resource relationships. This dependency introduces a potential single point of failure: even if business services are fully operational, access checks may fail, blocking critical functionality across the system. High-availability strategies or fallbacks are essential to mitigate this risk.



In some cases, systems embed relationship data directly into the request context. For example, an upstream service might enrich a JWT with project_ids or resource_links the user is connected to. This works well for performance, but it comes with risks, especially if those relationships are stale, incomplete, or forged. When implementing ReBAC in a distributed system, one approach is to embed relationship data (like "projectIds": [42, 51]) directly into the user's identity token (e.g., a JWT) or into the request context (see next snippet). This means the service doesn't need to look up

relationships at runtime. The app just checks what's already there.

```
{  
  "sub": "user-123",  
  "name": "Alice Johnson",  
  "roles": ["user"],  
  "project_ids": [42, 51, 88],  #A  
  "teams": ["team-finance", "team-europe"],  #A  
  "exp": 1717603200  
}
```

That can be very fast, because you're skipping database or graph lookups. However, it comes with some real risks:

The relationships embedded in the token may no longer reflect the current reality. For example, Alice was part of Project 42 yesterday, but she was removed today. If her token still says she belongs to Project 42, she can still access its resources until the token expires or is reissued. This is called a *stale relationship*.

The data in the token might be simplified or truncated to reduce token size or complexity. Maybe the system only includes a user's direct project memberships in the token, but not the ones inherited through team or organization relationships. As a result, some permissions may not be correctly enforced. This is called an *incomplete relationship*.

An attacker could forge or tamper with the data if a service blindly trusts the relationship data without validating the token signature or its source. Say a compromised frontend could send a fake token claiming the user belongs to a high-privilege group, and if no signature verification happens, the backend could wrongly grant access. This is called a *forged relationship*.

Some of these can be acceptable in high-trust environments, but in general, on-demand queries to a trusted relationship store offer stronger guarantees.

Tip

Design your system so services can easily ask, “Is this user connected to this resource in a valid way?” without having to know the full graph. Delegate that logic to a central service and treat it as part of your critical security

infrastructure.

ReBAC brings powerful flexibility, especially in systems with sharing, delegation, or nested teams, but it also introduces architectural complexity. Getting it right means investing in a reliable, queryable source of truth for relationships, and making sure every service knows how to use it.

17.2.3 Exercises

4. Why is authorization easier in monoliths?
5. What's the benefit of modular monoliths for access control?
6. Why should every service check access on its own?
7. What happens if services interpret roles or attributes differently?
8. Why do microservices use a central relationship store in ReBAC?
9. What's a big risk of using a central relationship store?
10. What's the problem with putting relationship data directly in tokens?

17.3 Answers to questions

1. What's the main difference between RBAC, ABAC, and ReBAC?
RBAC (Role-Based Access Control) gives access based on a user's role, like "admin" or "editor." It's simple: if you have the role, you get access. ABAC (Attribute-Based Access Control) goes further by checking user and resource details, like department, region, or time of request, making it more flexible. ReBAC (Relationship-Based Access Control) is about connections between entities. It looks at relationships like "Alice is part of Project X" or "this file is shared with Bob" and uses those to decide access.
2. Why does RBAC struggle in large systems?
In large systems, you often need more precise control than just basic roles. Over time, teams start creating lots of very specific roles, (like invoice_approver_eu or contract_editor_fr), to handle small differences in access. This leads to what's called role explosion: too many roles, unclear naming, and a system that becomes hard to manage or audit. It becomes difficult to know who has what access and why.
3. When would you choose ABAC over RBAC?
You'd choose ABAC when access decisions depend on more than just a

person's role. For example, if a user should access a file only if they are from the same department or region, RBAC can't handle that easily.

ABAC allows you to write rules like "user.department == document.department," making it ideal when you need more dynamic, context-aware decisions based on user or resource attributes.

4. Why is authorization easier in monoliths?

Authorization is easier in monoliths because all the parts of the system (like user info, business logic, and resources) live in the same codebase and usually the same process. That means you have all the data you need at your fingertips, and you can make access decisions without worrying about networking, consistency across services, or how to share identity data. It's much simpler than doing the same in a distributed system.

5. What's the benefit of modular monoliths for access control?

Modular monoliths simplify access control (RBAC/ABAC) by keeping all authorization logic in one deployable unit while still separating concerns by module. This allows consistent policy enforcement, easier auditing, and shared context for identity and permissions, without the complexity of distributed services.

6. Why should every service check access on its own?

In a microservices system, a request might pass through multiple services before it reaches the one holding the resource. If each service trusts the ones before it to do the authorization, a mistake in one place could let unauthorized access through. That's why every service needs to validate access independently. It's the only way to ensure consistent, secure enforcement in a system where you can't fully trust upstream checks.

7. What happens if services interpret roles or attributes differently?

If services don't agree on what roles or attributes mean, they may apply different access rules for the same user. For example, one service may treat the manager role as having full edit rights, while another sees it as view-only. This inconsistency leads to confusion and security issues. Users may be over-privileged in one service and underprivileged in another. Shared definitions and standards are key to avoiding this.

8. Why do microservices use a central relationship store in ReBAC?

In ReBAC, access decisions depend on relationships between users and resources—like ownership, membership, or delegation. But in microservices, no single service has all that relationship data. So each

service pushes its own piece (like “Bob is a member of Team A”) to a central relationship store. This store holds the full picture and lets services query it to make access decisions based on the complete relationship graph.

9. What’s a big risk of using a central relationship store?
The biggest risk is that it becomes a single point of failure. If the relationship store goes down or becomes slow, services that depend on it can’t make access decisions. Even if the rest of the app is healthy, you could end up blocking users from doing anything. To avoid this, you need high availability, caching, or fallback mechanisms.
10. What’s the problem with putting relationship data directly in tokens?
Adding relationship data like project IDs to the token makes authorization faster, but it comes with dangers. That data might be outdated (if someone was removed from a project after the token was issued), incomplete (missing indirect relationships), or even forged (if the token isn’t verified correctly). So while it improves speed, you trade off accuracy and security unless you handle it carefully.

17.4 Summary

- RBAC assigns access based on roles, making it easy to implement and widely supported, but it can lead to role explosion in larger systems.
- ABAC adds flexibility by evaluating user, resource, and environment attributes, enabling context-aware decisions but requiring consistent attribute definitions and data quality.
- ReBAC uses relationships between entities to determine access, allowing fine-grained permissions in collaborative or multi-tenant systems but needing a relationship graph or central store.
- Monoliths simplify authorization logic, making implementing and testing all three models locally easier. Authorization model implementation is even easier and more efficient when the monoliths are designed modular.
- In distributed systems, all services must agree on what roles, attributes, and relationships mean. If they don’t, access rules become inconsistent and security gaps can appear.
- Service-oriented systems demand each service enforce its own rules, making identity propagation and consistent interpretation of roles and

attributes essential.

- Cloud-native and serverless architectures offload part of the authorization logic to IAM policies, combining app-level decisions with infrastructure-enforced rules.
- In distributed environments, consistency and trust are key.
- Choosing between RBAC, ABAC, and ReBAC depends on the system's complexity and access patterns, and in many systems, a hybrid approach provides the best balance.

Appendix A. Installation and Setup

This appendix covers

- Setting up a Java development environment to use with the sample apps
- Where to get the sample code for the book

You can run all the sample applications and their dependencies on your laptop. This appendix tells you what you need to setup your laptop / workstation to run all the examples as you read the book. This book assumes you have access to a Windows, Linux, or MacOS machine that will be configured to run:

- Java 11 Development Kit
- Your Favorite Java IDE capable of working with maven projects (E.g. IntelliJ IDEA, Eclipse IDE)

A.1 Setting up Java Development Environment

The sample applications are either plain Maven projects or they are based on Spring Boot 2.3.x and Java 21 (with Maven as a build tool). Spring Boot is compatible with Java 8 but we have chosen to use Java 21 because it is a more recent long term supported release of Java at the time of writing.

A.1.1 Setup Java 11

The book samples have been tested with OpenJDK distributed by the AdoptOpenJDK project. You can download and install AdoptOpenJDK 21 from <https://adoptopenjdk.net/>. Other distributions of Java 21 should be 100% compatible with AdoptOpenJDK feel free to use those if you have them on your machine.

A.1.2 Setup A Java IDE

The sample projects use Apache Maven as the build system. You can compile the code and run it from the command line, or you can import it into your favorite IDE as a maven project. The sample have been tested with Eclipse and with IntelliJ IDEA. I recommend you use the most recent version stable release of Eclipse or IntelliJ IDEA to work with the code.

A.2 Obtaining the Book Samples

You can find the code samples for the book online

<https://github.com/Software-Security-For-Developers/software-security-for-developers>. There is a repository for samples from each chapter.