Compliments of vmware

With examples using Spring and Kubernetes

Securing Cloud Applications

Adib Saikali

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Securing Cloud Applications
With examples using Spring and Kubernetes

Version 3

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Securing Cloud Applications*

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

brief contents

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?

We will take a top-down approach to security starting with how to secure a single monolithic application, then a collection of microservices. 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

- Identifying how security roles & responsibilities are divided up between developers and everyone else in an organization implementing DevSecOps
- 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. My bank replaced my credit card twice in a five-year period due to data breaches at large retailers where I shopped.

We used to think that security vulnerabilities are primarily a software issue. However, hardware security vulnerabilities have been quite common in the past few years. Specter and Meltdown[1] reported in January 2018 allowed attackers to bypass the CPU hardware protection for memory access. In a cloud or multi-tenant environment specter and meltdown make it possible for one cloud tenant to see the memory of another tenant. The hardware walls that we depend on to keep workloads isolated 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 the collective responsibility of everyone building and running IT systems. From hardware engineers that design the processors in our phones to application developers building the ecommerce applications that keep our kitchens stocked with food. Regardless of your role in the IT world security is your responsibility.

[1] https://meltdownattack.com/

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

1.1 Security is a CEO level problem

The average cost of a data breach in 2020 is $3.92 million[2]. Some mega breaches such as Marriot Hotels leaked 500 million[3] customer records. Marriott took a $126 million charge to deal with the data breach. Equifax an American credit reporting agency spent 1.4 billion[4] dollars on cleanup costs associated with the 2017 data breach of 150 million personal credit histories.

A large security incident has the potential to be a company ending event. CEOs are quite concerned about the business impact of security breaches, so they are appointing 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 required 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 be able to 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. This means that as a developer you need to become familiar with new tools, processes, and practices used to implement DevSecOps.

A search for the term "computer security" on the amazon.com books section returns 40,000 results. It is easy to get lost in the details when learning computer security. The diagram below provides a map of the broad areas of application security.

[2] https://www.csoonline.com/article/3434601/what-is-the-cost-of-a-data-breach.html
[3] https://www.wsj.com/articles/marriott-take-126-million-charge-related-to-data-breach-11565040121
[4] https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473

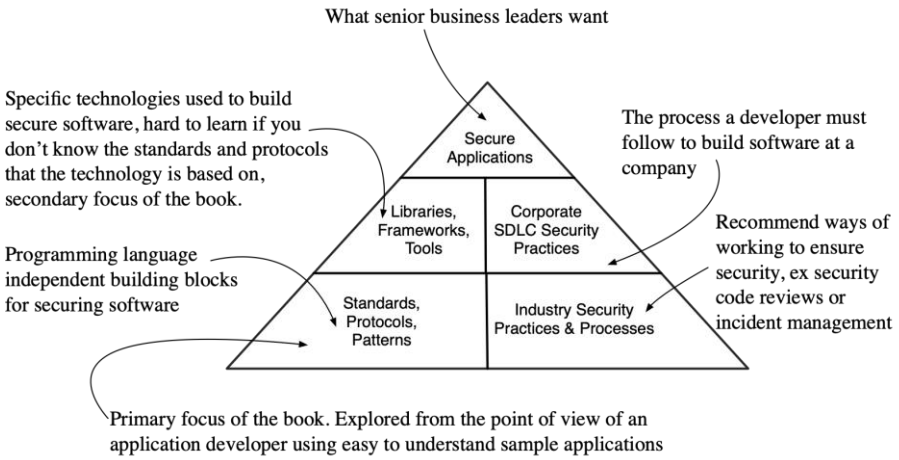


Figure 1.1. Layers at the top depend on the layers below them. All the layers are required to produce secure application. The standards, protocols, and patterns used to secure applications are the primary focus of this book, they are the foundation that you need to effectively use security libraries in your application

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. 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 implement industry standard protocols and patterns in a specific programming language. For example, Transport Layer Security (TLS) in the Java Standard Libraries or OpenID Connect (OIDC) in Spring Security. Naturally developers put a lot of effort into learning the security libraries they depend on. However, may developers find security libraries hard to learn and difficult to use.

The root cause of developer difficulties using security libraires is lack of knowledge about the underling 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 use and learn. 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 know the standards, protocols, and patterns you will find it easy to use security libraries and frameworks. Take the time to learn the underlying standard, protocols, and patterns.

This book focuses on teaching you the standards, protocols, and patterns implemented by the majority of application security libraries, and frameworks trough easy to understand use case based sample applications. The sample applications are implemented in Java using open-source frameworks such as Spring Security, Nimbus, Google Tink and others. So long as you can read Java code you will be able to learn the security standards, protocols, and patterns. For Java developers the book will give you a head start with these commonly used frameworks so you can more easily deep dive on the specific features that interest you using the frameworks online documentation.

Every company has a *Software Development Life Cycle* (SDLC) process that they follow for building applications. In mature companies the SDLC includes a set of security process and practices that developers are expected to follow in order to ensure that applications are secure. Companies build their SDLCs by customizing industry best practices for their needs. For example, an SDLC might be based on an agile process such as Scrum with security practices such added such as:

- Automated tooling to scan for security vulnerabilities in application code
- Security design review at the start of each sprit
- Security code review at the end of each sprint

Developers need to understand the problems that various security processes and practices are designed to solve. Understanding the problems makes it easier to correctly apply the practices and processes. This book provides you with the background required to work well in secure software development Lifecyle, but it does not teach practices or process. Checkout the book "Secure By Design" by Dan Bergh Johnsson, Daniel Deogun, Daniel Sawano if you are interested in learning more about how to design secure software.

The first step in the learning journey is to build a big picture mental model of application security use cases and the available approaches for solving them. In this chapter, we start the learning journey by analyzing two common security problems:

- Securing communication channel
- Securing application dependencies,

Examining how to secure communication channels and application dependencies enables us to make sense of types of skills an application developer needs to know about security and map out an effective approach for learning security skills.

In the next chapter we will analyze a common set of security problems encountered when building monolithic and microservice based application so that you can have a big picture understanding of the security technologies and standards that every developer should be familiar with in order to build secure cloud native applications.

Put on your boots, grab a strong coffee, we are about to explore the security mountain, it will take some effort, but we have made the trail as clear as possible, with good rest stops on the climb up the mountain. I promise you the view from the top of the mountain is breath taking and worth the sweat and effort you will put in.

1.2 Securing communication channels

Consider ACME Inc. a shoe retailer with 1000 physical stores in 5 countries and an online shopping application that customers use to buy shoes from anywhere in the world. Customers can shop for shoes on ACME's website or native mobile apps.

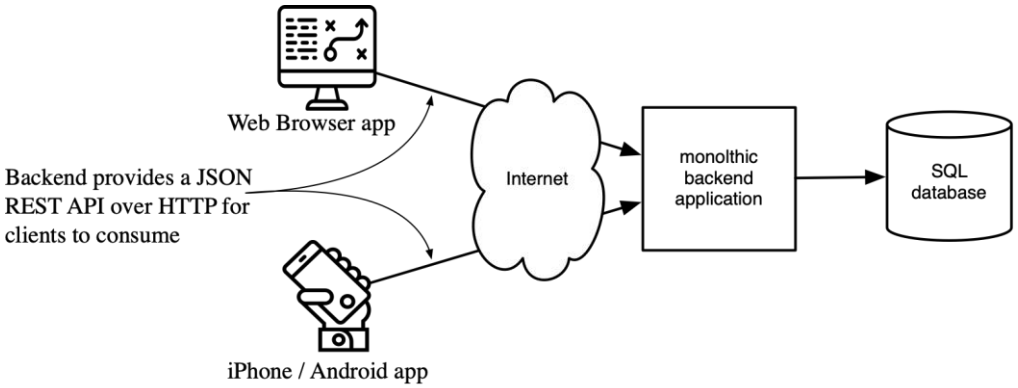


Figure 1.2 The high-level architecture of the ACME Inc. shopping applications. Three shopping applications: browser, iPhone, and Android, communicate with a monolithic backend using an HTTP JSON based REST API.

ACME's application architecture is typical for large enterprise applications. The frontend user interface consists of three separate applications: an HTML5 based web app, a Kotlin based Android app, and a Swift based iOS app. The backend exposes a REST API over HTTP using JSON as the data exchange format. The backend is a large application (1,000,000+ lines of code) running on an application server that process the HTTP requests and stores state in a SQL database.

As data is transmitted between a user's device and the application backend it passes through a number of untrusted networks such as the WIFI in a coffee shop where the user is enjoying a coffee and using the application on her mobile phone. Hackers have an opportunity to steal the data while it is in transit on untrusted networks connecting the user to the datacenter hosting the backend. Therefore, all communication between the application front end user interfaces and the backend must be encrypted with the industry standard Transport Layer Protocol (TLS).

The application backend accesses its database over a private data center network. Data center networks have many access controls and defense mechanisms. Network engineers segment corporate networks into zones that are separated from each other via firewalls and other network security devices. For example, load balancers that accept traffic from the internet are isolated into an untrusted red zone, while the application servers are put in a yellow zone, and the database servers are in a green zone.

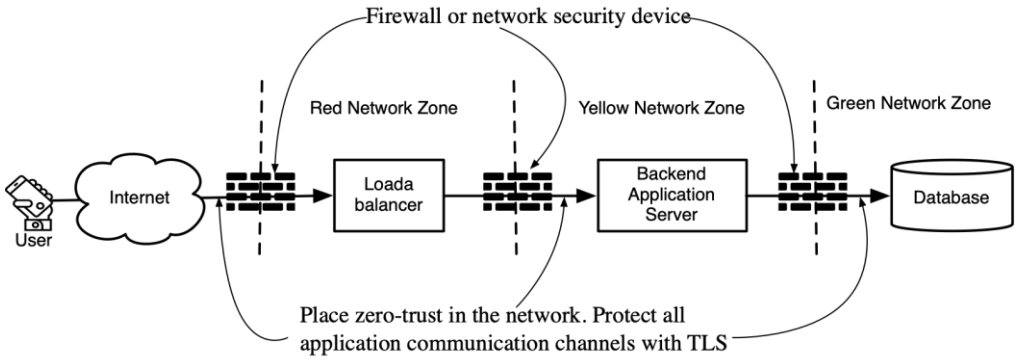


Figure 1.3 An example network segmentation. The red zone is for hosting publicly accessible servers such as load balancer. The yellow is zone is for hosting application servers, it cannot be accessed from the internet, but it can be access from the red zone. The green zone is for hosting data services and can be accessed from the yellow zone. Even in a highly segmented and zoned network it is a best practice to still use TLS for encrypting all application communication paths.

It is tempting to assume that there is no need to encrypt communications between the application backend and its database because the traffic is on a "trusted" internal network. Resist the temptation to trust the data center network for the following reasons.

- *Configuration errors*: security configuration errors on the network are possible especially in large corporate networks with hundreds and thousands of applications.
- *Disgruntled insider*: a company insider with legitimate network access might choose to attack the application.
- *Compromised application*: The "internal" corporate network has many applications on it, some are internal applications built by the enterprise, some are COTS applications purchased fam a variety of vendors. If a single applications on the internal network is comprised, then the whole network is compromised.

For example, Target[5] a large American retailer suffered a data breach of its point-of-sale system (POS) that affected 41 million consumers and cost $18.5 million in fines. Hacker breached[6] Target's POS by gaining access to the network with the credentials used to maintain the air conditioning system in Target stores.

It is a best practice to operate under the zero-trust networking model where you assume that the network is always untrusted. Treat the internal data center network with the same level of suspicion that you treat the internet. As an application developer it is important that you insist on TLS everywhere for any application network communications. Getting comfortable with the TLS protocol is a critical security skill for developers. Mastering TLS enables you to:

[5] https://www.usatoday.com/story/money/2017/05/23/target-pay-185m-2013-data-breach-affected-consumers/102063932/
[6] https://www.computerworld.com/article/2487425/target-breach-happened-because-of-a-basic-network-segmentation-error.html

- Write secure applications that meet corporate security standards.
- Quickly configure TLS in your code without spending hours searching blogs and stackoverflow.com for setup instructions.
- Debug connectivity issues caused by TLS configuration settings easily.

TLS is large complex protocol; whole books have been written about it. This book will demystify TLS, it will be covered from the point of view of an application developer. We don't assume you have any previous experience with TLS or cryptography so we will build all the foundational computer security concepts required to fully understand TLS. In order to understand TLS, you will need to understand the following security building blocks.

- Cryptographic Hash Functions
- Message Authentication Code (MAC)
- Hashed Message Authentication Code (HMAC)
- Advanced Encryption Standard (AES)
- Block encryption operating models
- Authenticated encryption
- RSA public key cryptosystem
- Elliptic curve public key cryptosystem
- Diffie-Helman key exchange
- X.509 digital certificates
- Certificate authorities and public key infrastructure

There is a tangled web of algorithms based on deep beautiful mathematics at the heart of TLS. You do not need to understand how these algorithms work or the math behind them, but you must understand what they do and how to configure them correctly in your applications.

To keep the book developer focused we will use a set of sample applications to explore the security building blocks. We will present sample applications written in Java using popular open-source Java libraries such as Spring, Google Tink, Nimbus and others. Running these applications will teach you what the security algorithms do and expose you to the various configuration options when using them. The code is written to demonstrate key security concepts, you will be able to run the code and learn the application security concepts even if you are not a Java developer.

Mastering TLS is a critical skill for an application developer. If you take the time to go through Part 2 & 3 of the book, run the sample applications, we promise that you will learn enough about TLS that you will never again get stuck debugging a TLS issue. You will always know what to look for to resolve the issue. Part 2 & 3 is organized in a step-by-step manner with each section building on top of the previous one, so we recommend you read it in the order it is presented.

> **BEST PRACTICE** Place zero-trust in all networks including internal "secure" networks. Use TLS everywhere for all application communication channels with everyone and everything.

Roles and Responsibilities

There are three organizational roles involved in securing communication channels: security engineers, network engineers, and developers. Security engineers provide guidance on how secure applications and networks. The guidance is typically published as a collection of corporate information security documents laying out the required configuration of TLS, network segmentations, and numerous other security related requirements. Security engineers perform audits and assessments of all levels of the IT stack to ensure compliance with corporate standards. Network engineers own the setup and management of the corporate networks in accordance with the guidance provided by the security engineers. Developers own writing code that follows the guidance provided by security engineers this means configuring TLS clients and servers to only use approved configurations. Developers provide network engineers with connectivity requirements so that they can configure the network to enable communication in the most secure manner between an application and its collaborators. As a developer mastering TLS makes you a better partner to the security and network engineers.

1.3 Securing application dependencies

Applications are built on top of hundreds of open-source libraries and propriety software components. For example, at time of writing 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 for 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[7] dollars on cleanup costs. The hack was caused[8] 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. The Equifax hack illustrates the importance of keeping all components in an application's software supply chain patched and up to date.

In 2018 a backdoor for stealing[9] Bitcoin was added to a popular JavaScript library called Event-Stream that averages 2 million weekly downloads from the NPM repository. The backdoor successfully attacked the Copay[10] Bitcoin wallet versions 5.0.2 to 5.1. 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 being vigilant against software supply chain attacks. Attacks against the software supply chain are increasing because they are extremely 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. For example, the `spring-boot-starter-data-jpa` dependency tree shown in the diagram below contains 45 .jar files from 15 separate open-source projects that must be validated.

[7] https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473
[8] https://www.zdnet.com/article/equifax-confirms-apache-struts-flaw-it-failed-to-patch-was-to-blame-for-data-breach/
[9] https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/
[10] https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/

Figure 1.4 the dependency tree for spring-boot-starter-data-jpa dependency includes 45 individuals .jar libraries from 15 separate open-source projects.

Even though the application does not contain code that calls methods in the *txw2* dependency txw2 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 other suppliers …etc. New security tools and processes must be built then adopted widely to secure the software supply chain.

At the time of writing the software industry is starting to tackle software supply chain security seriously. For example, On Mach 9 2021 the Linux Foundation announced sigstore[11] a non-profit project and service that open-source projects can use to improve the security of the

[11] https://sigstore.dev/

software supply chain. It will take a few years for the technologies and process to secure the software supply chain to be put in place industry wide.

1.3.1 Continuous automated dependency vulnerability detection and patching

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, build scripts and generated artifacts. The scanner compares the application's dependency versions against a database of known vulnerabilities. If a match is found the scanner alerts the development team.

The scanner should run on every code commit so that it can immediately alert the development team if the commit introduced 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 very 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. The ideal sequence of events for fixing the vulnerability is:

- CVE-2017-5638[12] is discovered and reported in the industry standard *Common Vulnerability and Exposers* (CVE) database with score of 10 which is highest possible score indicating a critical vulnerability that should be patched immediately.
- The vulnerability scanner vendor updates the scanner to look for CVE-2017-5638. Most scanning vendor are able to access the CVE databases before the public is notified about the CVE, so that when the CVE becomes public the scanner is already updated to detect the issue.
- The scanner automatically rescans your application determines that it is using Apache Struts version 2.3.25 which is vulnerable to CVE-2017-5638.
- The scanner modifies the maven pom.xml file for the application changing the Struts version to 2.2.32 which contains the fix for the vulnerability. It then opens a git pull request with the modified version, which triggers the application's continuous integration pipeline to build the application using the new version of Struts and run all the automated tests.
- The scanner alerts the application development and corporate security team about the issue.
- The development team reviews the changes proposed by the scanner's pull request, checks that are all tests are passing on the continuous integration server, merges the pull request and triggers a new release to production containing the vulnerability fix.

[12] https://nvd.nist.gov/vuln/detail/CVE-2017-5638

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

> **TIP** Rapid patching is only possible if it is easy to upgrade an application's dependency quickly. Sometimes developers use internal private interfaces from libraries to implement features. If you use internal APIs the application will get stuck on an old unsupported insecure version of the library. Recently, I ran into mission critical enterprise application that was 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 public published interfaces so that there is a documented path for upgrading to the next version whenever it gets released. Be kind to your future self and other developers that maintain the code that you are writing today by sticking to public 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[13] that you can turn on with the click of a button. However, there are many other vendors offering excellent dependency vulnerability scanners. The vendor list is constantly changing as start-ups release new products and existing companies get acquired.

As an application developer, it is important 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 can teach you the cryptographic primitives that these tools are built on. Part 2 of the book covers the foundational security technologies and standards used in all areas of security 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.

> **BEST PRACTICE** Use a dependency vulnerability scanner in all your applications, rapidly fix issues flagged by the vulnerability scanners. Stay up to date with advances in software supply chain security tools and processes and champion their use with your employer.

Roles and Responsibilities

An organization's specialist information security engineers' vet and asses third party libraries before they are used in applications. They also acquire dependency vulnerability scanners which are integrated into continuous delivery pipelines by application developers or DevOps teams.

[13] https://github.com/dependabot

1.4 DevSecOps

Most companies have three major silos in their IT departments: development, operations, and security. Traditionally these silos work independently, they interact by throwing stuff over the wall and blame each other for project delays, cost overruns and loss of productivity.

The information security silo writes security standard documents that operations and development teams are supposed to follow. Development teams complain that the security standards are restrictive, hard to understand and out of date with the latest software development practices and technologies. Operations teams complain that they have too many security related tasks to complete and are unable to keep up with patching and upgrading production systems.

Developers write applications then throw them over the wall to the security team to review and certify that the application meets corporate security standards and is approved for production release. Once the security review is complete the application gets tossed over the wall to the operations to deploy to production. The operations team complains that the apps are difficult to deploy and operate; they cannot keep up with the pace of deployment requests.

The business that is funding IT complains that feature requests are taking too long to implement and are too expensive, customers are unhappy. DevSecOps is a movement to break down the silos between development, operations, and security. The goal of DevSecOps is to reduce the time, effort, and cost of taking a business idea from concept to production while applying the best practices of security, operations and development. Companies are investing heavily in DevSecOps tools and transformation projects, such as:

- Setting up continuous automated application dependency vulnerability detection and patching, requiring the introduction of new tools and a change to existing IT processes.
- Building a secure Kubernetes based infrastructure to automate the rapid deployment and patching of applications and the underling infrastructure.
- Building automated integration test suites that can run on every commit and provide the confidence that an application deployed to production without manual testing.

DevSecOps is an organization wide team effort touching a lot of technologies and processes that are critical to securing all applications be they legacy monolithic applications or cloud native microservices. As an application developer this book teaches you many concepts, standards and technologies that are used to implement DevSecOps. However, the book is not about DevSecOps, it is about the security standards and technologies that all developers should be familiar with.

1.5 Information security roles and responsibilities

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 that is most relevant to your needs for writing secure applications. The diagram below shows a continuum of security related skills that can help you understand what you need to know and where to focus your efforts for learning security.

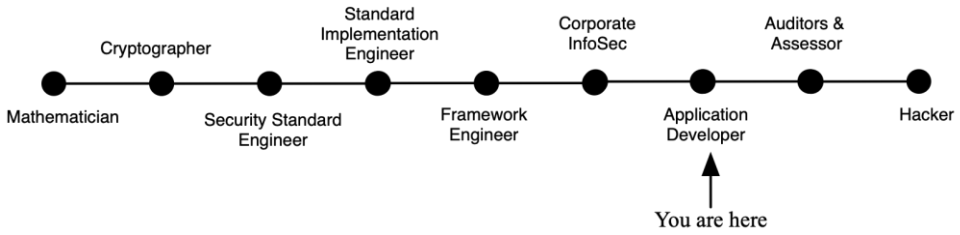


Figure 1.5 the spectrum of technical roles involved in computer security roles and responsibilities

Mathematicians produce the foundations upon which computer security rests. For example, elliptic curves are used to implement public key cryptography frequently used 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, secure commination 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 algorithm 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 examples of standards you will frequently encounter as developer. It is important 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 of searching blog and stackoverflow.com
- Debug security issues and configurations quickly

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

Framework engineers implement libraries to accelerate application development in a specific programming language. Framework developers focus on common use cases that applications that 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 effectively build applications. The book uses the Spring Framework ecosystem to demonstrate how to implement a variety of common security uses 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 the technical background to quickly understand corporate security standards. A key goal of the book is to help you became a better partner to your InfoSec team.

Auditors and assessors evaluate the design and implementation of applications to ensure that 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 the 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.

1.6 Security technologies every developer should know

Application developer should be very comfortable 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. TLS is covered in part 3 of the book.

TLS is built on top of foundational cryptography algorithms and standards. Two critical standards for developers to understand are:

- *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.
- *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 above standards depend on a solid understanding of cryptographic primitives. We will cover cryptography from a developer friendly perspective in part 2 of the book. 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.

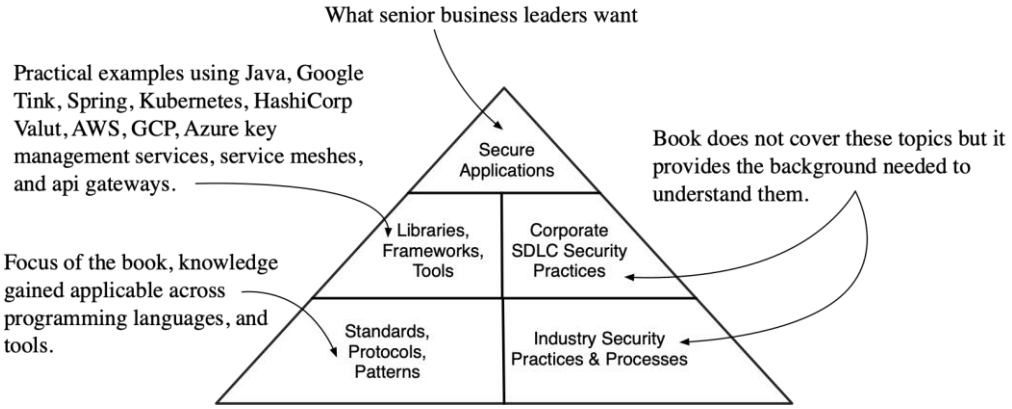


Figure 1.6 The book teaches you the standards, protocols, and patterns for building secure applications using practical sample applications. The goal is to set you up for success on your security learning journey by teaching you foundational technologies every developer should know.

1.7 Summary

- Never trust the network treat internal "secure" networks with the same level of trust as the internet. Secure all application communication channels using Transport Layer Security (TLS) protocol.
- Transport Layer Security (TLS) is a key foundational technology that every developer should know really well. This book will provide will teach everything you need to know about TLS to use it correctly and quickly debug issues.
- Mastering TLS requires understanding a lot of technical background that will be developed in part 2 & 3 of this book.
- Software supply chain attacks are increasing because they are extremely effective. Use a dependency vulnerability scanner in all your applications, rapidly fix issues flagged by the vulnerability scanners. Stay up to date with advances in software supply chain security tools and processes and champion their use with your employer.
- The goal of DevSecOps is to reduce the time, effort, and cost of taking a business idea from concept to production while applying the best practices of security, operations and development. Companies are investing heavily in DevSecOps transformations.
- Security is the collective responsibility of everyone working in IT including developers. Improving your security skills makes you more valuable to your employer and enables you to resolve security issues quickly.
- This chapter covered the security problems common to both monolithic application and microservice applications, the next chapter will focus on the security challenges faced by microservice based applications.

2

Foundational Security technologies

This chapter covers

- Analyzing customer, employee and partner preferences for logging into applications along with the standards that enable secure user authentication
- Identify the technologies for securing sensitive application credentials
- Analyzing problems faced securing service-to-service calls and the technologies available to secure the service-to-service call graph
- Finalizing the list of security technologies every developer should know

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, and running the application securely in a cloud native environment such as Kubernetes. In the last chapter we discussed the technology choices for securing communication channels using TLS, in this chapter we will examine the technology landscape for solving the rest of the security problems common to microservices and monoliths.

In a microservice based application a single user request is processed by multiple microservices that call each other. Securing the service-to-service call chain is super important. In this chapter, we will examine the challenges associated with securing service-to-service calls and we will identify common solutions and patterns.

At the end of this chapter, you will have a comprehensive list of all the technologies and patterns that you should be familiar with as a developer. This list enables you to develop a personal learning plan to master the key technologies and patterns used to build cloud native applications.

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. 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 password 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 account on un-related 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[14] 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.

Consider ACME Inc. the shoe retailer with 1000 physical stores in 5 countries and an online shopping application that we discussed in the last chapter. 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 in order to maximize sales. Returning customers should be able to quickly login to place an order. New customers should be able to quickly provision a new account so that they can have a fast checkout process when they return in the future to shop for more shoes.

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

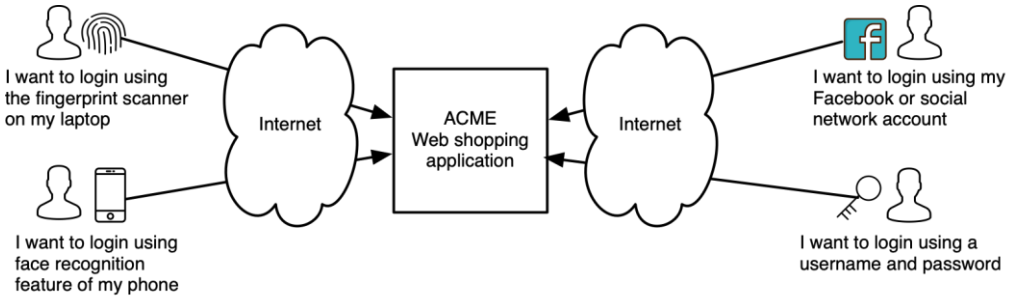


Figure 2.1 Customer authentication preferences. Some customers want to use the biometric authentication features on their devices and phones such as fingerprint scanners, and face recognition. Some customers want to be able to login using their existing accounts with large online service providers such 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.

There are three 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 for enabling users to use existing accounts across online services. Part 5 of the book covers 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 login into it using the biometric capabilities of device they are using 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 as well as 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 part 5 of the book.
- *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 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.

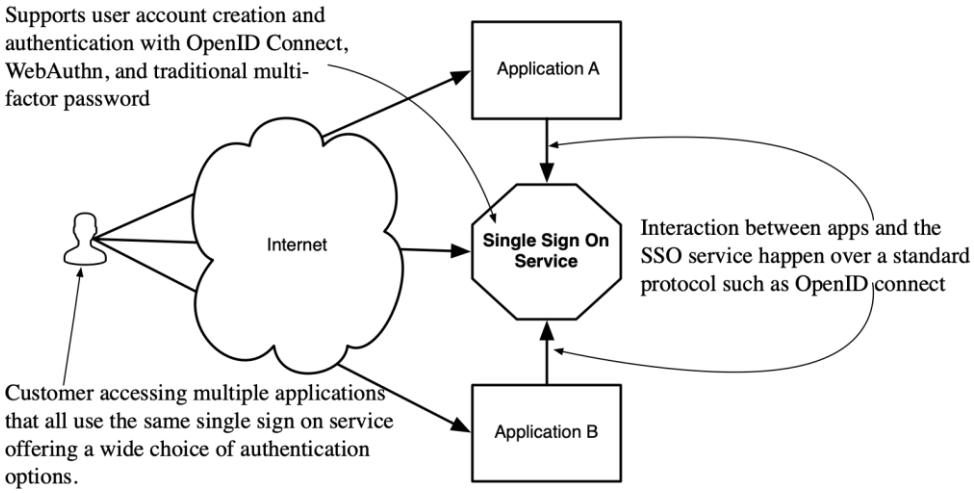


Figure 2.2 Single Sign On (SSO) Service handles user account creation and authentication. Multiple applications can use the same SSO service simplifying security for application developers. The SSO service implement support authentication using OAuth2, OpenID Connect, WebAuthn, and multi-factor password-based authentication.

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 Keycloack, 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 developer getting comfortable with OAuth2, OpenID Connect, WebAuth2, and the technologies for multi-factor password-based authentication provides with a complete toolbox for all your authentication needs in 2021. 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 way to deepen your understanding of SSO technologies.

2.1.2 Employee authentication

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 using 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 service based on how they are setup in Active Directory. When an employee leaves ACME inc. their external SaaS based account should be deprovisioned 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 customer to login with customer accounts and employees to login with their Active Directory accounts?

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

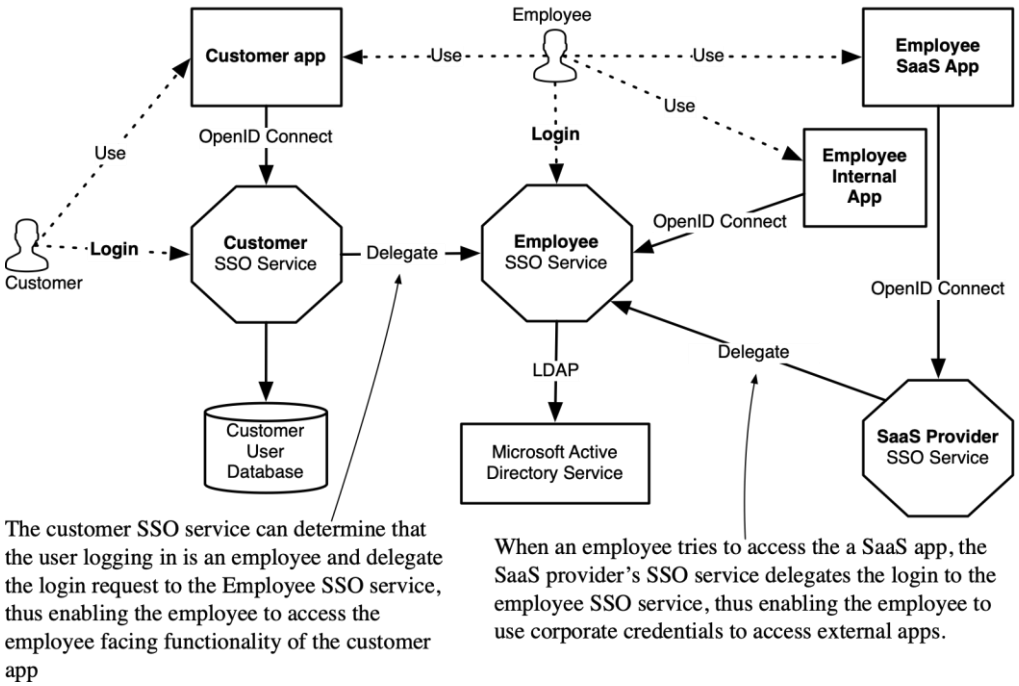


Figure 2.3 A corporate Sigle Sign On (SSO) service makes it possible for employees to access customer facing apps, internal employee only apps, and external SaaS apps using a single set of credentials

Every company with more than a handful of employees must implement access controls to corporate systems. When employees are hired, they must be granted access, when they 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) so you can use it 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 very simple to implement. If a partner employee is fired or quits, ACME's IT staff must be notified in order 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 extremely effective.

For example, consider the following steps that hackers can use to break into ACME Inc.'s cloud infrastructure. Using a LinkedIn search hackers identify Joe Smith as the lead cloud DevOps engineer on ACME Inc's online retail team and that he can be reached at jsmith@acme.com. Through some online sleuthing the hackers determine 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 is 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 that is a replica of the Google login screen. Joe does not notice that the URL in 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 the 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 2.4 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 out a monolithic application into a Single Sign On (SSO) service is the recommend approach. 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 enables 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 remember.

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

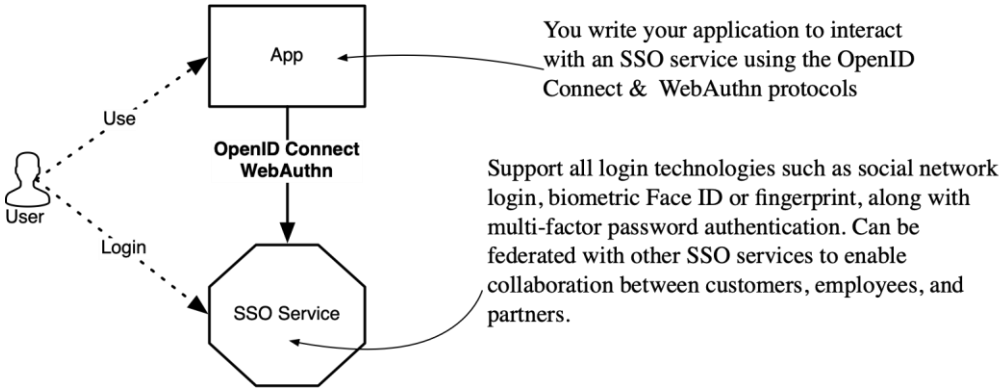


Figure 2.5 Externalize application authentication into an SSO service that applications can access using OpenID Connect and WebAuthn. If you know OpenID Connect and WebAuthn you can use all modern SSO services.

OpenID Connect and WebAuthn are supported via high quality libraries and frameworks in all programming languages. Part 5 of the book provides you with 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 a well-maintained open source SSO service rather than building your own. Keep your SSO service patched and up to date.

Roles and Responsibilities

An organization's specialist information security engineers make the decision on which SSO service can be used inside a company. They take care of setting the SSO service and selecting which configurations of OpenID Connect and WebAuthn they want to enable. As a developer you will need to add OpenID Connect and WebAuthn libraries to your application so that it can interact with the SSO service. If you understand the protocols you will find it very easy to use a programming language specific library, so focus on learning the protocols before you learn how to a specific implementation.

2.1.6 Exercises

1. Why are passwords terrible for security and user experience?
2. What protocol makes it possible to login 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 need to access database servers, message brokers, email servers, internal and external APIs to implement the application functionality. For example, ACME's online shopping application depends on the services shown in the diagram below.

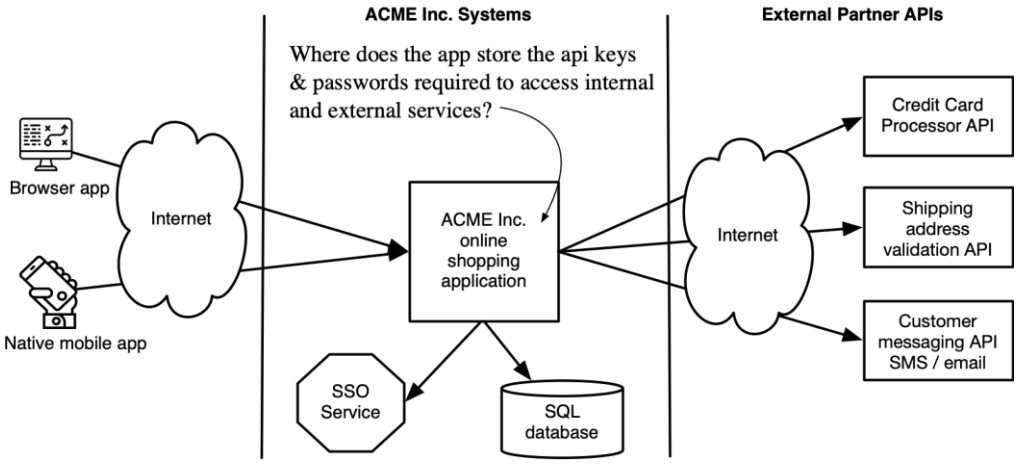


Figure 2.6 Applications depend on internal and external services that require passwords, and API keys to use. The credentials to access services are extremely sensitive and must be protected. How can an application store and access sensitive credentials it needs to operate?

When a customer buys 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 is coming from ACME's systems. If hacker gets access to the payment API key, they can cause a lot of 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 1 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 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 10 servers requires a copy of the configuration file on each server. It is easy for an update process to fail resulting in 2 out of 10 server 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 of a fine-grained audit trail required for investigating 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.

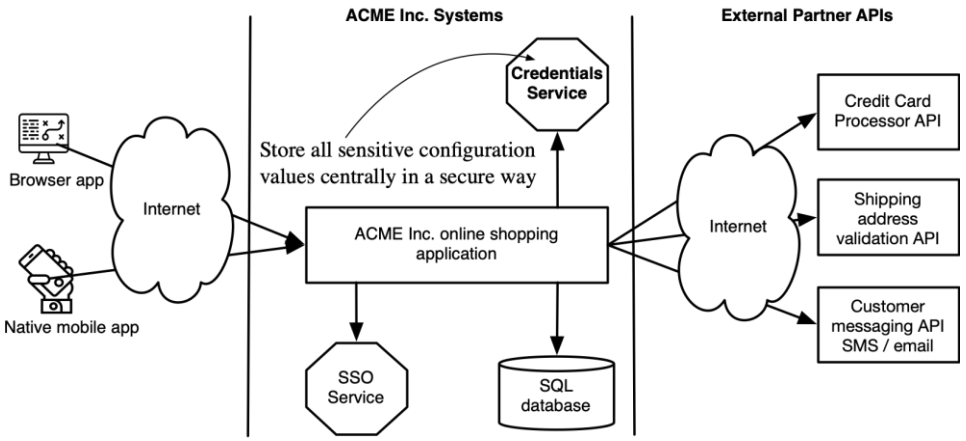


Figure 2.7 A credential vault acts as a centralized store of all sensitive configuration values such as passwords, API key, digital certificates, and other secrets that the application needs to access at runtime.

Putting all application secrets in a centralized credentials service is a great alternative to using configuration files. The primary benefits for 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 a credential is updated in the credential service applications can be notified about the updated value enabling applications to use new values without needing to restart the application.
- *Simplified credential rotation*. Credential services store metadata about credentials such as credential 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 the secrets they hold are well protected against even the most determined attackers.

Do not write you own credential service. Implementing and maintaining a credential service require very deep security expertise, it is very easy to make mistakes that cause catastrophic security failures. There are three common approaches to get access to a credential store.

- *Cloud provider credentials service*. Public cloud providers offers centralized credentials management services for applications running on the public cloud to use. For example, an application running on Azure can you the Azure Key Vault service to the manage all sensitive credentials. The book cover AWS Key Management Service, Azure Key Vault, GCP Key Management Service in part 4.
- *Container platform credentials service*. If the application is running on a container platform it can use the platform's built in credential service. Applications running on Cloud Foundry can use CredHub, a credential store built into the Cloud Foundry platform. Kubernetes provides a very basic 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, there is no industry standard protocol or API for accessing a credential service. Every credential service product has its own proprietary API and client libraries that you will need to 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 making the request. Thus, we have a bootstrapping problem:

- How does the application obtain the secret it needs to access the credential service?
- Where does the application store the secret it needs 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 difficult problem that can only be solved if it is baked into the platform running the application. Part 3 of the book covers the *Secure Production Identify Framework For Everyone* (SPIFFE) where you will learn how SPIFFE can bootstrap trust for applications running on Kubernetes.

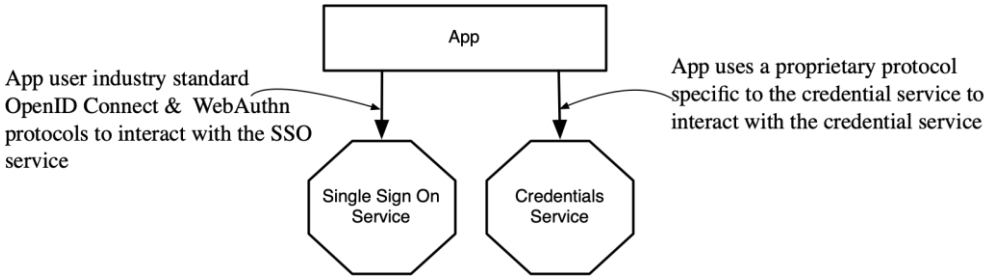


Figure 2.8 A credential management service and a single sign on service are two foundational components for cloud native application security. As a developer you can use the OpenID Connect and WebAuthn protocols to interact with all modern SSO services. However, for credential services there is no industry 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 will teach the patterns for handling secrets securely and teach 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 a well-maintained open-source credential management service.

Roles and Responsibilities

An organization's specialist information security engineers make the decision on which credentials service can be used inside a company. They take care of setting the credential service and providing guidance on how to use the service. As an application developer you need to have a solid understanding of how credential services, which in turns depends on your understanding of foundational security standards and protocols that we cover in part 2 of the book.

2.2.1 Exercises

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

2.3 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 shown below with potential microservices highlighted.

Figure 2.9 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. The diagram below shows how the product page microservice is composed out of other microservices.

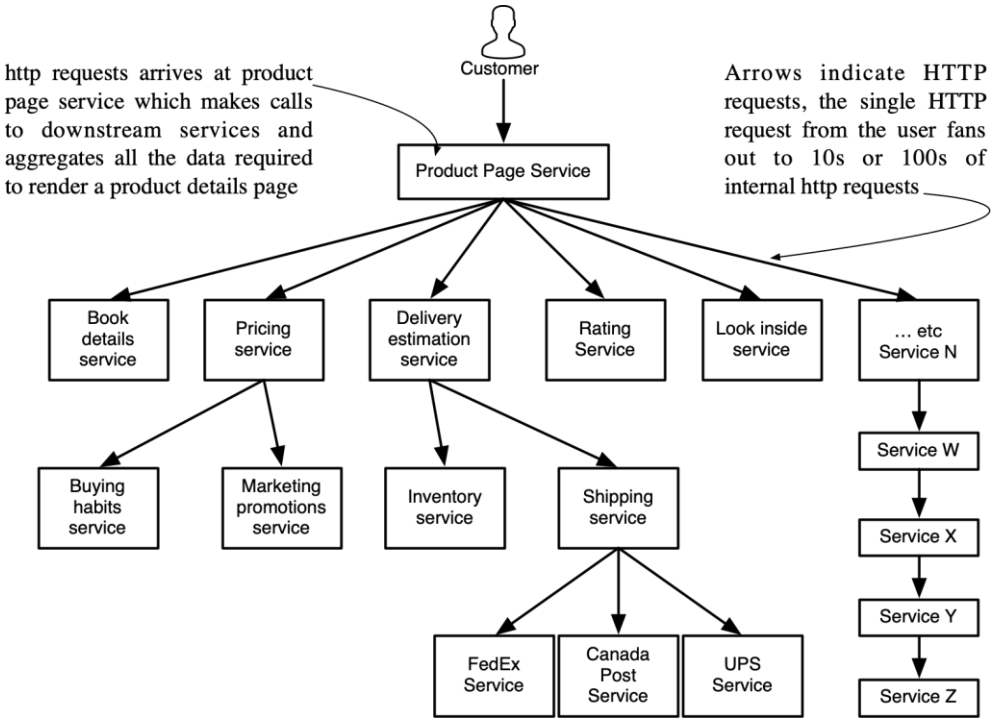


Figure 2.10 an http request to the product page microservice service fans out through multiple layers of supporting microservices, this is what we mean by the call service-to-service call graph. Organizations can have hundreds of microservices that interact with each other.

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 by 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 other services to consume. 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 making a purchase. It has a JavaScript library for rendering the book pages and navigation. The business logic ensures that the user can only read a few pages of the book 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 service that call each other as we have seen in the discussion above. Organizationally, each microservice is owned by a cross-functional team that is responsible for every aspect of the microservice, including requirements, design, development, testing, deployment, maintenance, and operations. Organizing teams around microservices enables rapid evolution of features and functionality.

> **TIP** microservice 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, the pros and cons of the approach. If you are looking for a review of microservices architecture checkout "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.

A single request to a microservice can fanout to become multiple requests. The fanout introduces two problems. First how can user identity be propagated from one microservice to another. Second how can a microservice determine the identity of the calling service. These two problems are illustrated in the diagram below.

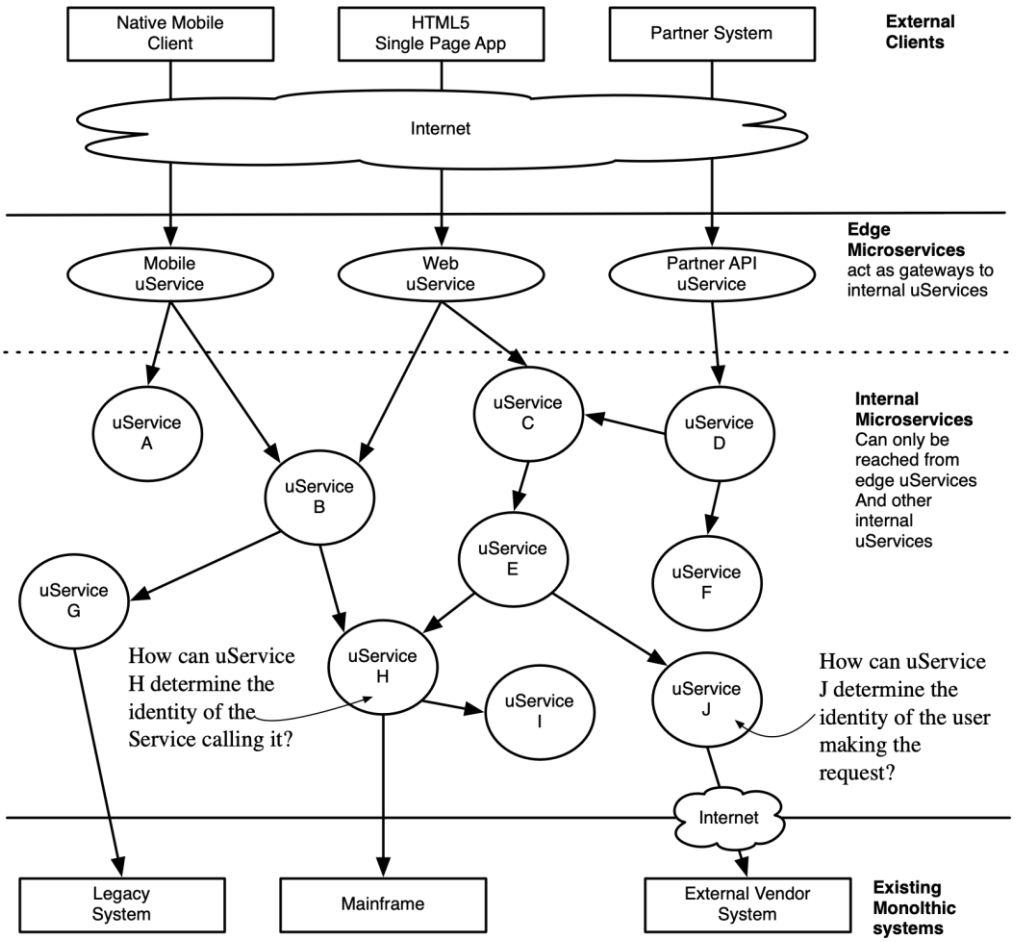


Figure 2.11 A generic call graph showing edge microservices that receive user requests then call internal microservices to handle the request. How can a microservice determine the identity of the user that initiated the request? How can a microservice determine the identity of the service that called it?

2.3.1 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
- Synchronous call using an RPC protocol such as gRPC or Thrift
- Asynchronous request-reply using a message broker such as RabbitMQ
- Asynchronous event notification using message broker such as Kafka

The edge service making the call might be written in Java while the service receiving the call might be written in JavaScript. We need a way to propagate user identity through a service-to-service call chain using different network protocols and programming languages as shown in the diagram below.

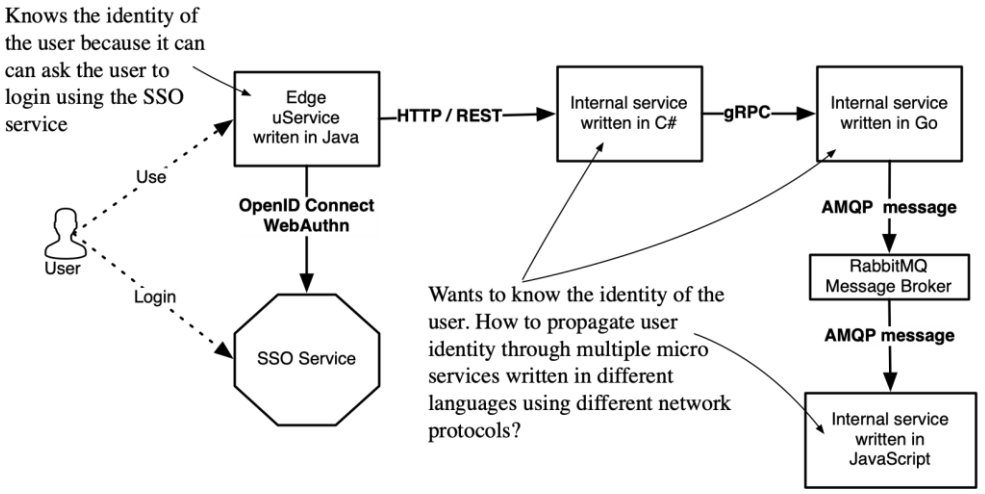


Figure 2.12 The edge service uses OpenID Connect or WebAuthn protocols to determine the identity of the user making the request. It must then propagate the user identity over HTTP to the service written in C# which must propagate the user identity over gRPC to the service written in Go which must propagate the user identity over AMQP to the service written in JavaScript.

There is a no standard way to solve the user identity propagation problem. You will need to 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, JOSE suite of standards, X.509 certificates, OAuth2, OpenID Connect, Service Mesh, API gateway. Be patient there is a lot of complex technology to digest and understand before we can solve this problem.

2.3.2 Determining service identity

Some microservice do not care about the identity of the user that 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 user id of the customer browsing the product catalog.

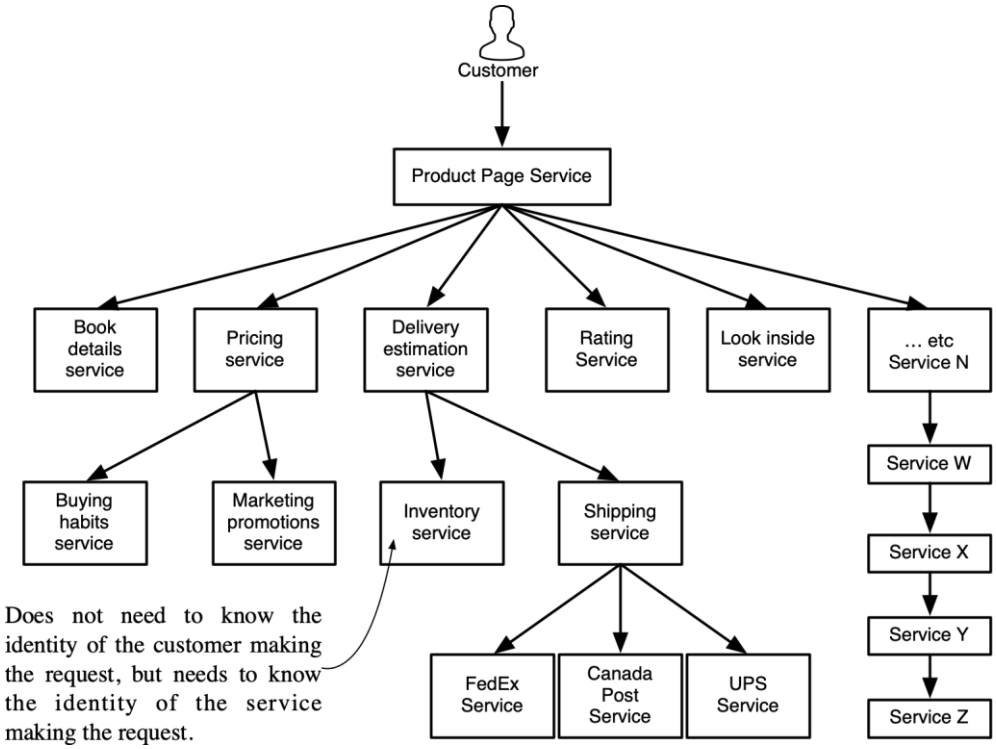


Figure 2.13 some service like inventory do not need to know the identity of the customer viewing a product page to return the current inventory level.

The inventory service needs to know the identity of the service making the request 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

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

- *API keys*
- Mutual TLS (mTLS) authentication

In the API key approach, the microservice expects the caller to include an HTTP header or a 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. The microservice, can determine the identity of the caller by checking the field is 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.

Roles and Responsibilities

Defining and configuring a secure environment for applications to run in is a joint responsibility between Information security and operations team. For example, deploying and maintaining a Kubernetes cluster with a service mesh installed and configured. Developers must be familiar with the patterns for securing the service-to-service call chain so that they can implement the patterns using the technologies available in the environment where the application runs. For example, using the service mesh to simplify the implementation of mutual TLS communication between service. This book show you the common patterns using samples implemented in Java, Spring running on top Kubernetes.

2.3.3 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 use 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?

2.4 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 to deploy and run. Developers did not need to know how to run applications in production and operators did not need to know 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.

Containerizing a custom-built application is the responsibility of the development team that coded the application. 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 you 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 the 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 show 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. Lean the best practices for writing secure Kubernetes deployment manifests and follow them.

Roles and Responsibilities

DevOps platform engineers are responsible for creating and managing secure Kubernetes clusters in accordance with corporate security standards set by corporate information security engineers. As an application developer you are responsible for containerizing your application securely and writing the Kubernetes manifests to run the application in a secure manner on Kubernetes. This book will show you the best practices for securely containerizing applications and running them on Kubernetes.

2.4.1 Exercises

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

2.5 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 above 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

We will cover cryptography from a developer friendly perspective in part 2 of the book. The goal of this book is to teach developers complex security technologies is 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.

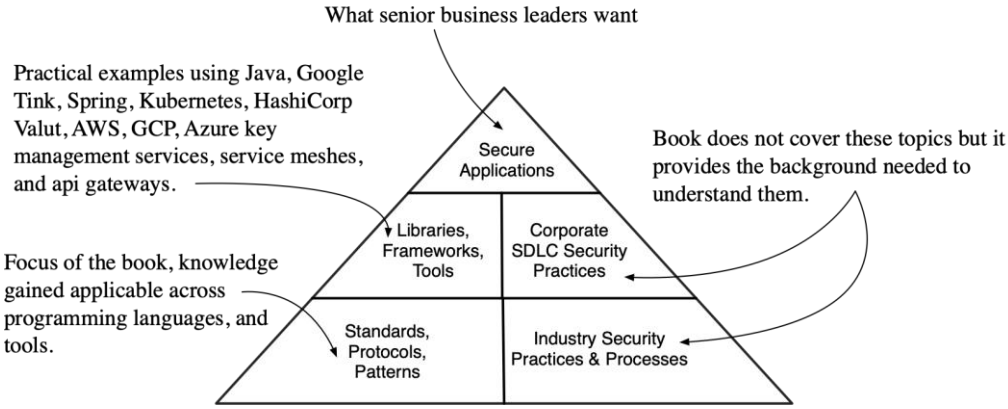


Figure 2.14 The book teaches you the standards, protocols, and patterns for building secure applications using practical sample applications. The goal is to set you up for success on your security learning journey by teaching you foundational technologies every developer should know.

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.

2.5.1 Exercises

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

2.6 Summary

- Passwords are terrible for usability. Reduce the number of passwords your users have to deal with by leveraging a Single Sign On (Service) that implements OAuth2, OpenID Connect, and WebAuthn.
- OAuth2 and OpenID Connect enable users to login using accounts they have already provisioned with online service providers such as Google, Facebook.
- WebAuthn enables users to login using biometric features of phones such as facial recognition or fingerprint scanners. It also works with phishing resistant physical security keys.
- Always externalize user authentication into an SSO service since it simplifies application coding effort, increases overall security.
- Always store sensitive application credentials in a credential service such as HashiCorp Vault or the cloud provider's key management service.
- Securing the service-to-service call chain is the most complex topic that we will cover in this book. You will need a solid foundation in all the pre-requisites technologies including cryptography, service mesh, API gateways, and a variety of patterns and conventions.
- Two aspects of securing the service-to-service call chain are: propagating user identity through the service call chain and establishing service identity.
- Securely containerizing application and running them on Kubernetes is an important skill for developer to possess.
- Be patient there is a lot of complex security technology that is book will guide through.

2.7 Exercise Answers

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

   Long complex passwords are hard to remember. There are numerous online services that require password. Most users don't use a password manager, so they end up using weak insecure passwords, or they reuse the same password multiple times. Storing password securely is hard and expensive so apps should avoid storing password and instead rely on a Single Sign On Service.

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

   The WebAuthn protocol makes it possible to login using biometrics such a phone's facial recognition chip, thumbprint scanner, or phishing resistant physical security keys.

3. **What is OpenID Connect used for?**

   OpenID Connect (OIDC) is an industry standard protocol implemented by the majority of SSO services. It is a universal API for interacting with SSO services.

4. **What is a phishing attack?**

   A phishing attack tricks user into entering their password and other credentials into a fake hacker created site that looks just like the site the user is used to using.

5. **Which authentication technology can protect against phishing attacks?**

   Physical security keys that can validate the site that the user is logging into can default phishing attacks. WebAuthn makes it possible to use physical security keys in your application.

6. **What capabilities does an SSO service provide?**

   An application can delegate user authentication to the SSO service so that the application does not have to store any passwords. The SSO service can provide password less login with WebAuthn protocol and social login with online service providers such as Login with Facebook. The SSO service can federate with other SSO services making it possible for an application written by one organization to trust the users of another organization.

7. **Should every application use an SSO service?**

   Yes, because it reduces effort and increases security, even old apps can benefit from being refactored to use an SSO service.

8. **Where should application secrets be stored?**

   Application secretes should always be stored in an application credentials storage service.

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

   Configure files are hard to secure, difficult to audit, make credential rotation difficult, and are hard to keep in sync across multiple machines.

10. **What are the benefits of using a credential storage service?**

    One source of truth eliminating configuration drift, simplified credential rotation, comprehensive audit logs, hardware security module support, simplifies the implementation of DevSecOps process.

11. **Should all applications use a credential storage service?**

    Yes, all apps should use a credential storage service because it is more secure and makes DevSecOps process much easier to implement. the application better.

12. **Is there an industry standard protocol or API that can be used to access a credential service?**

    At the time of writing there is no industry standard API or protocol to interact with a credential storage service, you have to reply on the providers implementation specific APIs and libraries.

13. **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 recieve requests from other microservices.

14. **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 a 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.

15. **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.

16. **Is there an industry standard solution for solving the use 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.

17. **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.

18. **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.

19. **What technology can help simplify the implementation of mutual TLS between services?**

    Service mesh such as Istio can make mTLS connectivity between microservices simple.

20. **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.

Part 2

Cryptography foundations

This part of the book is a practical introduction to cryptography for application developers. Computer security is built on a foundation of cryptographic algorithms that form the building blocks of higher-level protocols such as Transport Layer Security (TLS), OAuth2 OpenID Connect (OIDC) … etc. If you want to write code against security libraries that implement these protocols, you will need to understand the protocols. To understand the protocols, you will need to understand the following cryptography primitives:

- Cryptographic hash function
- Message Authentication Code (MAC)
- Hashed Message Authentication Code (HMAC)
- Symmetric Key Encryption
- Authenticated Encryption
- Authenticated Encryption with Associated Data (AEAD)
- Public key encryption
- Key exchange protocol

You will learn the cryptography concepts in the list above through sample applications that make use of the following algorithms and industry standards:

- Standard Hash Algorithm (SHA-2, SHA-3)
- Advanced Encryption Standard (AES)
- RSA public key crypto systems
- Elliptic Curve Cryptography (ECC) crypto system
- Diffie-Helman Key Exchange using ECC cryptography
- JSON Object Signing and Encryption (JOSE) suite of standards
    - JSON Web Algorithm (JWA)
    - JSON Web Key (JWK)
    - JSON Web Signature (JWS)
    - JSON Web Encryption (JWE)

- JSON Web Token (JWT)

As an application developer it is important that you understand what cryptography algorithms do and how to configure them correctly. The best way to learn about the algorithms and how to configure them is through sample applications that you can study and run. The sample applications are written in Java. Each application comes with a GitHub repository that you can use to run the application. The samples use the following Java libraries.

- *Spring Boot*, because it is one the most popular frameworks in the Java ecosystem.
- *Java Cryptography Architecture (JCA)*, because it is the official cryptography API that is part of the Java Runtime Environment (JRE).
- *Nimbus*[1], because it is a well-known implementation of the JOSE suite of standards, and it is used by the Spring Security Framework.
- *Google Tink*[2]*,* because it offers a developer friendly cryptography API designed to eliminate common cryptography misconfiguration bugs. Tink has implementations in Java, C++, Objective-C, Go, Python, JavaScript, TypeScript making Tink a good choice for a variety of projects. Google uses Tink in its online applications so the Tink is used at scale. Tink is actively maintained by Google security engineers making it a safe choice to use in your applications.

If you can read Java code, you will be able to follow along and learn the cryptography concepts you need to understand as a developer. The sample applications are optimized for educational value so that can fit in printed book pages and be accessible to the widest possible audience. There are no equations or mathematics anywhere in the book just sample applications that are easy for developers to run and understand.

[1] https://connect2id.com/products/nimbus-oauth-openid-connect-sdk
[2] https://github.com/google/tink

3

Message Integrity and Authentication

This chapter covers

- Guaranteeing data integrity using the Secure Hash Algorithm (SHA)
- Ensuring data integrity and sender authenticity using a Hashed Message Authentication Code (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.

No matter what programming language you write code in, or which cloud provider you deploy your application on, cryptographic algorithms are the foundational security building blocks. Terse documentation and mysterious error message 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.

Suppose you want to add a social login button to your application such as "Login with Google". You will need to use the OpenID Connect and OAuth2 protocols. 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 will need to understand Hashed Message Authentication Code (HMAC). To understand HMAC, you need to understand the concept of cryptographic hash functions and Message Authentication Code (MAC).

We will build two applications in this chapter. 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 Hashed Message Authentication Code (HMAC) to determine who created a file and prove that the file was not tampered with.

3.1 The goals of cryptography

Consider ACME Inc. a shoe manufacturer. Customers shop for shoes on ACME's ecommerce website and pay for orders using a credit card. For every order placed ACME's ecommerce system makes a call to the credit card API offered by a payment processing company.

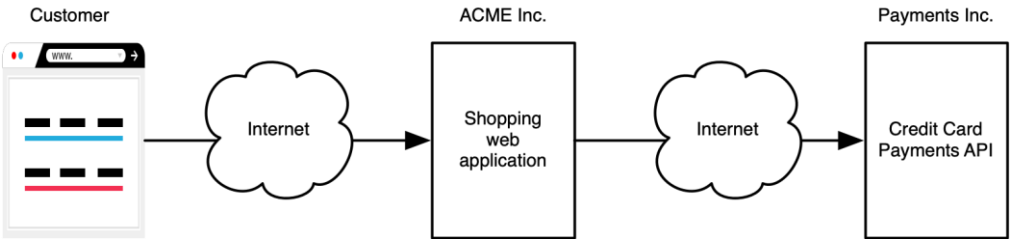


Figure 3.1 A typical ecommerce high level architecture. Customers place orders using the ACME Inc. web applications which in turn makes calls to the Payments Inc. credit card API to collect payments from customers while processing an order.

Cryptography is used to secure the HTTP requests from the user's web browser to the ecommerce website and from the ecommerce 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
- authentication
- confidentiality
- non-repudiation

Integrity ensures that data is not tampered with during transmission or storage. An HTTP request from a customer's browser to the ecommerce system can be alerted 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. Ensuring message integrity during transmission or storage is one of the fundamental problems that cryptography solves.

Authentication in the context of cryptography means that the recipient of a message 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?

TIP Authentication can refer to user authentication or message authentication. User authentication typically means asking a user 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 hacker cannot steal credit card details if they are able to intercept network communications. If an employee of ACME's 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 that 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.

Thinking about 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 3.1 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

It will take us several chapters to learn the algorithms in the previous table. 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, the rest of this chapter will teach you how to guarantee data integrity, and authentication.

3.2 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 check 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`. The following diagram illustrates this workflow.

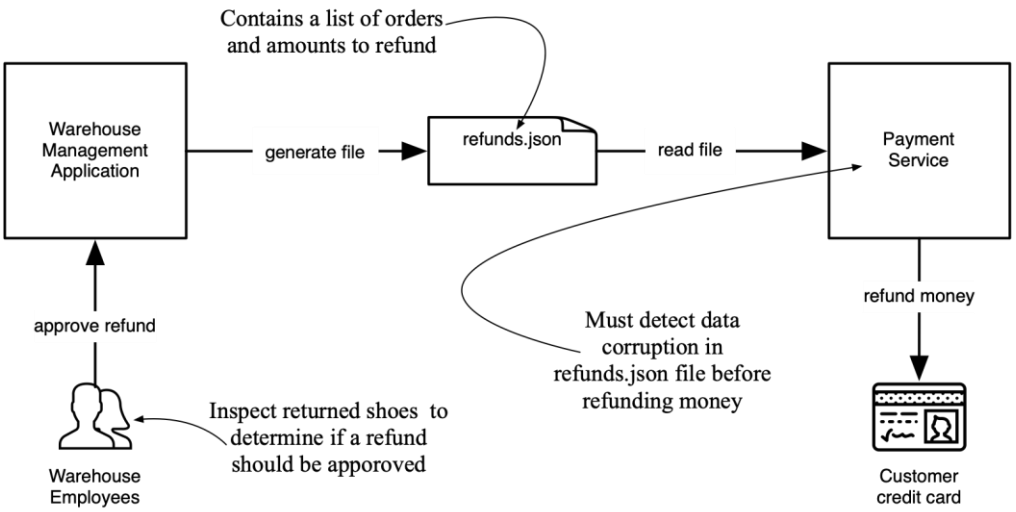


Figure 3.2 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service refunds customer credit cards for the amount specified in the refunds.json file.

Listing 3.1 Example refunds.json file produced by the warehouse application

```
[ {
  "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.

> **TIP** Chapters 3,4,5,6 and 7 make use of 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 https://github.com/securing-cloud-applications/. 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.

3.2.1 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. The following table shows example inputs and outputs from the SHA-256 hash function.

Table 3.2 Hash values using SHA-256 for inputs of varying sizes

Input	SHA-256 hash value represented as hexadecimal number
1-byte lowercase "a"	ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
1-byte uppercase "A"	559aead08264d5795d3909718cdd05abd49572e84fe55590eef31a88a08fdffd
2.6GB ubuntu.iso file	b45165ed3cd437b9ffad02a2aad22a4ddc69162470e2622982889ce5826f6e3d

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 functions that have mathematical properties that make them suitable for use in computer security. Two primary properties of a cryptographic hash function are:

- *one-way property*: Given the output of the hash function it is not feasible to determine the original input that the hash function executed on.
- *Collision resistance*: different input values produce completely different output values. It should not be possible to find two inputs that produce the same hash value.

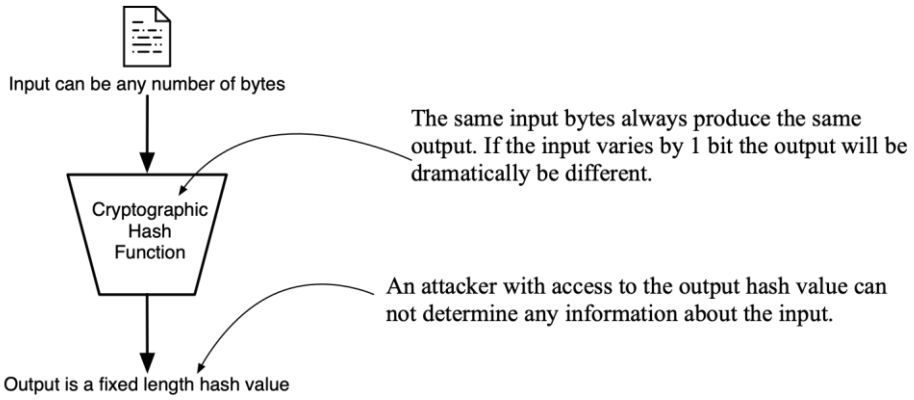


Figure 3.3 Think of hash function as a blender that takes input of different sizes to produce output of the same size. Different input values produce completely different output values. Attackers with access to the hash value are unable to determine anything about the input.

A lot of mathematics is used to design and assess the security of cryptographic hash functions. It takes years of effort, a lot of scrutiny and peer review by cryptographers to reach 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[15] security researchers found a way to generate collisions with the widely used SHA-1 hash function. As result SHA-1 is no longer considered secure.

TIP The secure algorithms of today 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 family of functions is widely supported by all programming language environments including Java.

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 variety of memory, speed and security trade-offs. At the time of writing 256-bit is the minimum setting that is considered secure. Before you write code using a cryptographic hash function research the current recommended output size. For example, www.keylength.com aggregates recommendations from various government organizations on the minimum key sizes that should be used for a variety of cryptographic algorithms.

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

TIP if you have implemented a hashCode() method on a Java object you might be wondering how the Java hashCode() method is different or similar 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.

3.2.2 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:

- Compute the SHA-256 hash value on the contents of the `refunds.json` file.
- 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.

Table 3.3 Pseudo code for using a cryptographic hash function to validate file integrity

Warehouse Management Application	Payments Service

`saveRefunds(Path location, bytes[] content)`
`  hashValue = CryptoUtils.sha256(content)`
`  writeFile(location, content)`
`  writeFile(location + ".sha256", hashValue)`
	`isRefundsFileValid(Path location)`
	`  content = readFile(location)`
	`  expected = CryptoUtils.sha256(content)`
	`  actual = readFile(location + ".sha256")`
	`  if( expected == actual) return true`
	`  else return false`

The following diagram shows the generic validation process graphically.

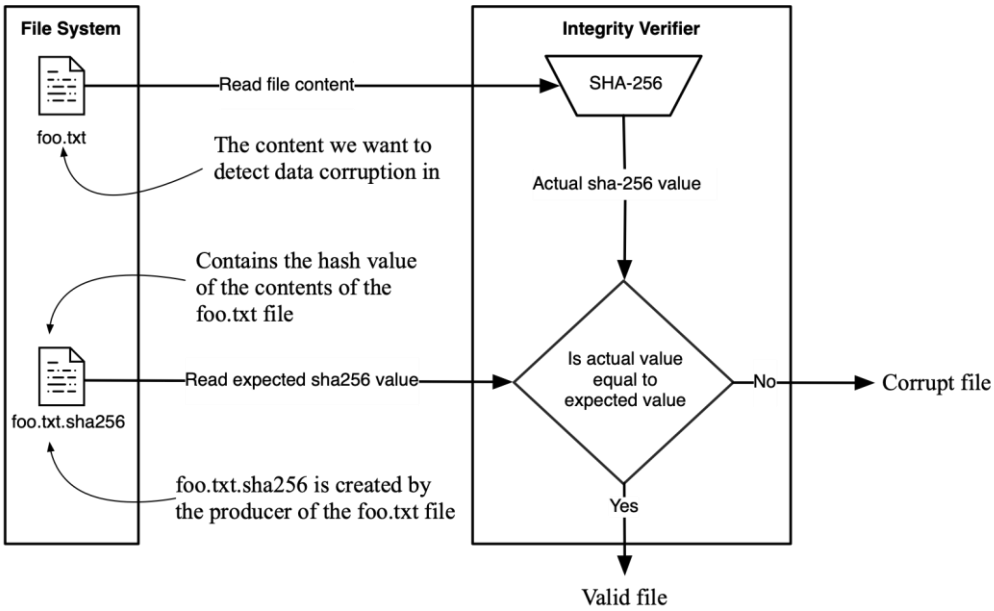


Figure 3.4 Detecting data integrity violation using a cryptographic hash function. The foo.txt producer computes the hash value of the content then writes it to foo.txt.sha256. A consumer computes the SHA-256 value of the contents of foo.txt then compares the computed value to the expected value in foo.txt.sha256. 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 to process an invalid file.

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 in this chapter. 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, that are 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. The following screen shot shows the Spring Framework's webmvc module files stored on the Maven Central repository.

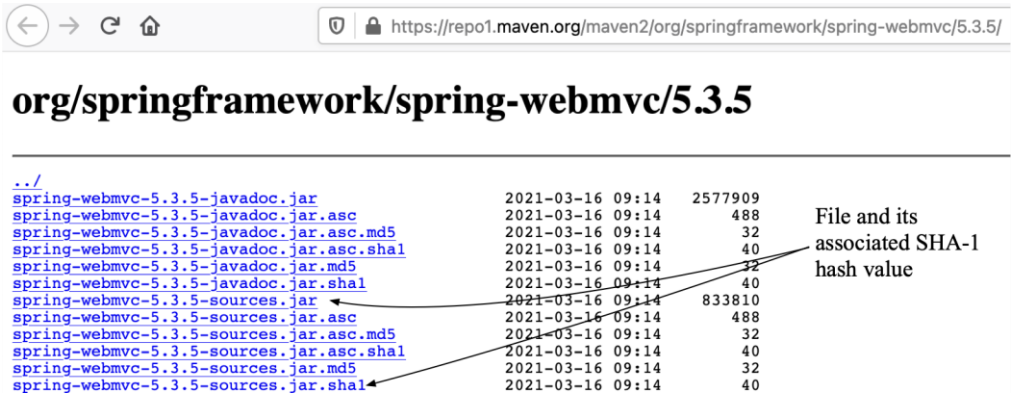


Figure 3.5 Files on maven central. Notice the sha1 and md5 files that contain hashed values of the corresponding artifacts. MD5 and SHA-1 are insecure, but they are still used by maven central for backward computability reasons.

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 screen shot a set of files that end in `md5` and `sha1` extensions for each of the files stored in maven central. At the time of writing 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[16] 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 contents of the commit. Below 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
```

#A The SHA-1 hash value representing the git commit

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 repository:

[16] https://issues.sonatype.org/browse/MVNCENTRAL-5276

- Computes the SHA-1 of the file being added
- Check to see if the SHA-1 value already exists in the repository.
    - If the SHA-1 exists in the 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.
    - 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 it's 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[17].

3.2.3 Design for hash function change

The maven central and git examples discussed earlier demonstrate the need for applications to upgrade the hash functions they are using. If you are designing a system that uses a cryptographic hash function assume that it is a matter of time before you have to change the hash function.

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

> **WARNNIG** attackers will set version fields to known to old values in order 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 not possible 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 a variety of applications, they are used in all the security protocol 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. You can use a cryptographic hash functions to accomplish the following cryptography goals.

[17] https://lwn.net/Articles/811068/

Table 3.4 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 of the Java Cryptography Architecture and Extensions which we will be using in many samples throughout the book.

3.2.4 Exercises

1. What are the two properties that differentiae cryptographic hash functions from regular hash functions?
2. What are the differences between SHA, SHA-1, and SHA-3?
3. Suppose you are working an insurance claims submission application. Customers install the app on their phone, take photos of their damaged property and submit a claim. The backend for the mobile application is using PostgreSQL as the database, and the AWS S3 object storage service. Think through the following design questions:
   a) How would use SHA-256 as part of implementing the photo management feature?
   b) How can the application track in PostgreSQL where images are stored in S3?
   c) Write down the `CREATE table` statement for the `photos` table that can track where photos are stored in S3.
   d) If a file is updated in S3 without the app's knowledge can the app detect that the file was changed?

3.3 Java cryptography architecture and extensions

The *Java Cryptography Architecture* (JCA) and the *Java Cryptography Extensions* (JCE) provide cryptography support in Java. Java's cryptography support is spread over the JCA and JCE libraries due to USA government cryptography export controls 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 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.

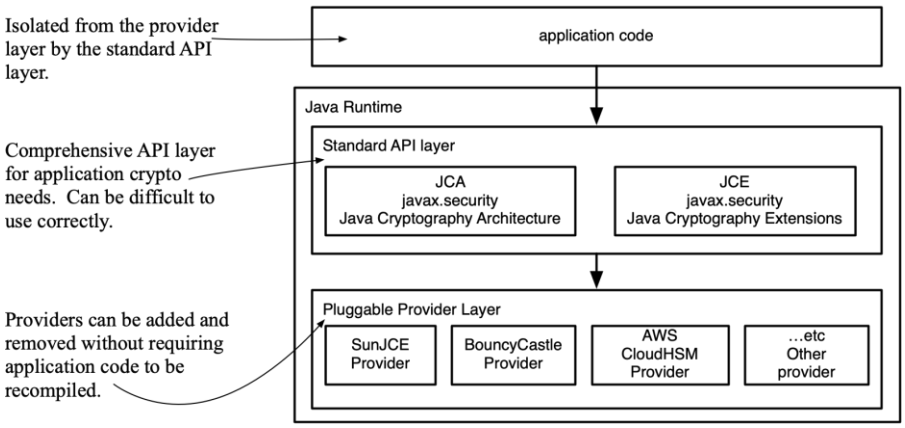


Figure 3.6 Application code calls the standard JCA and JCE APIs making the application code independent of the implementation of the crypto algorithm. The JCA and JCE layer routes call to a provider's implementation. Java 11 based OpenJDK ships with 13 providers out of the box.

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

The SunJCE[18] 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 extensions .jar was available for download as an addon to the Java Runtime Environment to enable strong cryptography for Java users in countries friendly to the American government.

The Bouncy Castle[19] JCE provider was created by Australian developers who are not subject to USA export control laws and therefore able to offer strong cryptography support to all Java users. Bouncy Castle was also optimized to run in embedded devices that have 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 ship with unlimited strength encryption.

A *Hardware Security Module* (HSM) implements cryptography algorithms in hardware in order 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[20] 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 11 OpenJDK

[18] https://www.oracle.com/java/technologies/javase-jce8-downloads.html
[19] https://www.bouncycastle.org/
[20] https://docs.aws.amazon.com/cloudhsm/latest/userguide/java-library-install.html

bundles 13 security providers by default. Developers access algorithms using names that are standardized and documented[21] 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. The following code listing shows a simple utility function for computing the SHA-256 value of a byte array returning the result as a hexadecimal encoded string.

Listing 3.2 Utility class for working with the Java Cryptography APIs

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import org.springframework.security.crypto.codec.Hex;

public class CryptoUtils {
  public static String sha256(byte[] input) {
    try {
      MessageDigest sha256 = MessageDigest.getInstance("SHA-256");    #A
      byte[] hash = sha256.digest(input);    #B
      return String.valueOf(Hex.encode(hash));    #C
    } catch (NoSuchAlgorithmException e) {  #D
      throw new RuntimeException(e);
    }
  }
}
```

#A obtain hash function instance using standard name
#B compute the hash value
#C convert the computed hash value into a hex string
#D error handling if the requested algorithm is not supported by java version running the app

The `java.security.MessageDigest` class provides access to variety of industry standard hashing algorithms. The MD2, MD5, and SHA-1 are insecure hashing algorithms that are 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.

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

TIP The JCA and JCE are powerful generic low-level APIs but are not developer friendly, it is easy to accidentally misconfigure or misuse them. The reset of this chapter and the next 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[22]. 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 libraires. Chapter 7 will teach you how to use the Java version of Tink.

3.3.1 Google and Amazon JCA providers

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, there are low level optimization techniques which 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[23]

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 that were previously too resource-intensive to deliver."[24] .

Google Conscrypt is a JCA provider implemented on top of BoringSSL[25] 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, it is best to start with the default JCA provider implementations that ship with OpenJDK. You can switch providers once the cost savings from a non-default provider outweighs the complexity of adding it your deployment pipeline.

3.3.2 Exercises

What changes do you have to make the code in listing 3.2 to change it from returning a SHA-256 hash code to a SHA-3 with 256 bits of output?

[22] https://github.com/google/tink
[23] https://conscrypt.org/
[24] https://aws.amazon.com/blogs/opensource/introducing-amazon-corretto-crypto-provider-accp/
[25] https://boringssl.googlesource.com/boringssl/

3.4 Implementing message integrity in Java

It has taken us a few pages to learn about the concept of a cryptographic hash function, 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 to get 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 refunds file is sent to the payment service for processing as shown the following diagram.

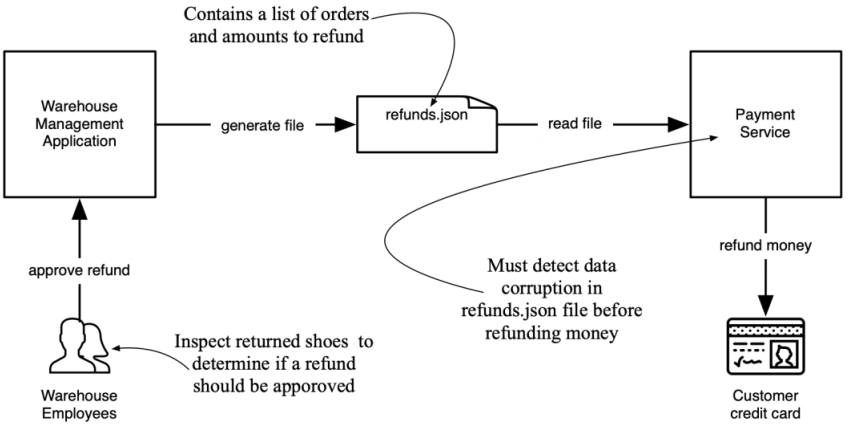


Figure 3.7 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service issues refunds to customer credit cards for the amount specified in the refunds.json file.

The sample application discussed in this section implements the ACME Inc. data corruption detection scenario. You can find the code on GitHub at https://github.com/securing-cloud-applications/crypto-hash. The GitHub repo contains detailed instructions explaining how to run the application on a developer laptop. I highly recommend you run the sample application to deepen your understanding of the topic. There are three projects in the sample repo:

- *util*, a shared library containing the `CryptoUtils` class with helper functions for working with the Java crypto libraries.
- *warehouse*, a Spring Boot 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 otherwise an exception is thrown.

The `util` project contains the code for the `CryptoUtils` class. `CryptoUtils` was explained in the previous section, it is reproduced below so you can have all the code for the scenario in one place.

Listing 3.3 Example refunds.json file produced by the warehouse application

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import org.springframework.security.crypto.codec.Hex;

public class CryptoUtils {
  public static String sha256(byte[] input) {
    try {
      MessageDigest sha256 = MessageDigest.getInstance("SHA-256");    #A
      byte[] hash = sha256.digest(input);    #B
      return String.valueOf(Hex.encode(hash));    #C
    } catch (NoSuchAlgorithmException e) {  #D
      throw new RuntimeException(e);
    }
  }
}
```

#A obtain hash function instance using standard name
#B compute the hash value
#C convert the computed hash value into a hex string
#D error handling if the requested algorithm is not supported by java version running the app

The following `Refund` class is used by the warehouse application and the payments service as the Java representation of the orders to be refunded.

Listing 3.4 Example

```
public class Refund {
  private String orderId;
  private BigDecimal amount;

  public Refund(String orderId, BigDecimal amount) {
    this.orderId = orderId;
    this.amount = amount;
  }

  // getters and setters not shown
}
```

The `Refund` class in kept to a bare minimum so we can demonstrate the cryptographic hash function concepts, without making the sample application complex. The payment application can use the `orderId` filed to determine which credit card to refund. The Jackson library is used to convert an instance of the `Refund` class into a JSON object like the one below.

Listing 3.5 refunds.json file produced by the warehouse application

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

The warehouse Spring Boot application is a command line application hardcoded to generate the JSON in the previous listing. The following listing show the code for the warehouse application.

Listing 3.6 Warehouse Spring Boot application

```
@SpringBootApplication
public class WarehouseApplication implements CommandLineRunner {

  @Autowired private RefundGenerationService refundGenerationService;

  @Value("${refundsPath:data/refunds.json}")
  private String refundsPath;

  @Override
  public void run(String... args) throws Exception {
    var refunds = List.of(                                  #A
            new Refund("12345", BigDecimal.valueOf(500)),   #A
            new Refund("56789", BigDecimal.valueOf(250)));  #A

    refundGenerationService.generateReport(Path.of(refundsPath), refunds);
  }

  public static void main(String[] args) {
    SpringApplication.run(WarehouseApplication.class, args);
  }
}
```

#A hardcoded sample data

The warehouse application uses the `RefundGenerationService` to create the `refunds.json` and `refunds.json.sha256` files. It accepts a path to write the files to and a list of refunds to save. It uses the CryptoUtils class from listing 3.3 to compute the SHA-256 value.

Listing 3.7 RefundGenerationService creates the refunds.json & refunds.json.sha256 file

```
@Component
public class RefundGenerationService {

  public void generateReport(Path refundsFile, List<Refund> refunds) {
    try {
      String refundsJson = JsonUtils.toJson(refunds);   #A
      Files.writeString(refundsFile, refundsJson);      #A

      String hashValue = CryptoUtils.sha256(refundsJson.getBytes());   #B

      String shaFilename = refundsFile.getFileName() + ".sha256";    #C
      Path hashFile = refundsFile.resolveSibling(shaFilename);       #C
      Files.writeString(hashFile, hashValue);                        #C

    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A convert the refund list into a refunds.json file
#B compute the SHA-256 hash value of the generated json
#C write the hash value into refunds.json.sha256 file

The `refunds.json.sha256` file contains the SHA-256 hash value `b730a046b3bb308bb2713353945c79be457e2420e4e81ea2d2bd6151e0128576` computed over the following JSON content

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

The Spring Boot payments service application reads the files generated by the warehouse application as shown in the following listing.

Listing 3.8 Payments Spring Boot Application

```
@SpringBootApplication
public class PaymentsApplication implements CommandLineRunner {

  @Autowired private PaymentService paymentService;      #A

  @Value("${refundsPath:data/refunds.json}")
  private String refundsPath;

  @Override
  public void run(String... args) throws Exception {
    paymentService.processRefunds(Path.of(refundsPath));   #A
  }

  public static void main(String[] args) {
    SpringApplication.run(PaymentsApplication.class, args);
  }
}
```

#A PaymentService contains the logic to process the refunds.json file

The payment application is a simple command line interface rather than a API because we want to keep the sample as simple as possible. The `PaymentService` class shown in the following listing implements the logic to:

- Read the `refunds.json` file.
- Compute the actual SHA-256 value of the `refunds.json` file.
- Read the expected SHA-256 value from the `refunds.json.sha256` file.
- Compare the expected and actual values, if they don't match throw an expection otherwise process the refunds.

Listing 3.9 PaymentService validates the refunds.json before processing refunds

```
@Component
public class PaymentService {

  public void processRefunds(Path refundsFile) {
    try {
      byte[] refunds = Files.readAllBytes(refundsFile);    #A
      String actualHash = CryptoUtils.sha256(refunds);  #A

      String shaFilename = refundsFile.getFileName() + ".sha256";    #B
      Path hashFile = refundsFile.resolveSibling(shaFilename);       #B
      String exceptedHash = Files.readString(hashFile);              #B

      if (!actualHash.equals(exceptedHash)) {     #C
        throw new CorruptRefundFileException();   #C
      }                                           #C

      System.out.println("Issuing Refund to");
      System.out.println(Files.readString(refundsFile));
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
#A read the refunds.json file and compute the actual SHA-256 hash value
#B read the expected hash value from refunds.json.sha256
#C raise an error if the expected and actual hash values don't match
```

By computing the SHA-256 hash of the `refunds.json` file and comparing it with expected hash stored in the `refunds.json.sha256` file the payment service is able to 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.

> **TIP** we are going to use the project structure explained in this section through the rest of part 2. Make sure to browse the full source code at https://github.com/securing-cloud-applications/crypto-hash there are detailed instruction for how to run the sample apps. Getting familiar with the sample code in the repo now will make the rest of this part of the book easier to understand.

3.5 Message Authentication Code (MAC)

In the ACME Inc. refunds scenario implemented in the previous section we were able to detect 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.

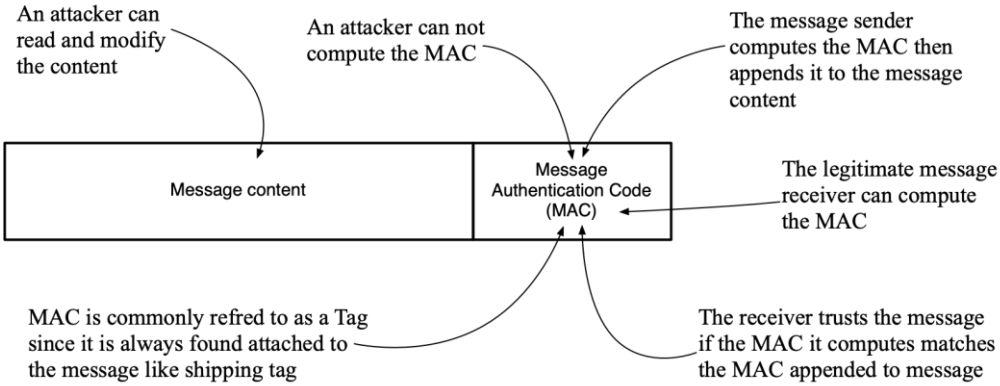


Figure 3.8 Message Authentication Code (MAC) is data added to the end of a message that can be used by the receiver of a message to identify who sent the message and that the message was not tampered with.

The sender, receiver, and a hacker can compute the cryptographic hash of a message 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 show in the following diagram.

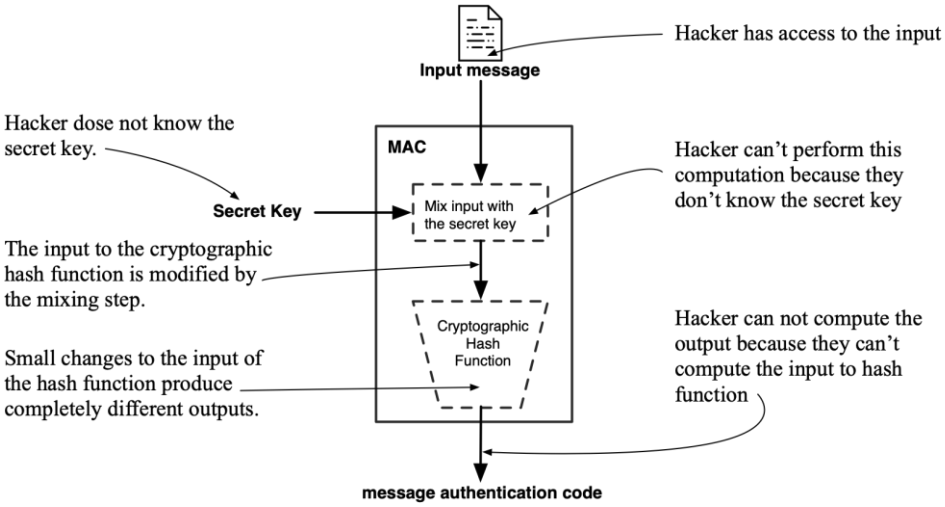


Figure 3.9 Computing the Message Authentication Code (MAC) requires both the secret key and the input message. When the hacker modifies the input, they cannot compute the MAC because they don't know the secret key that is part of the computation. Since they can't compute the MAC they can't fool the reciever with a fake message body.

There are many algorithms for implementing a MAC. Building the MAC using a cryptographic hash function as illustrated in the previous diagram is a popular approach. However, you have to be careful in 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 not secure, while using SHA-3 is secure. HMAC described in the following section is a secure way to build a MAC out a hash function.

3.5.1 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 || is used to indicate concatenation. For example, "abc" || "123" results in the string "abc123". You will see the || 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 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.

JAVA SUPPORT FOR HMAC

Java 11 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 below 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 returning the result as a hexadecimal string.

Listing 3.8 Computing an HMAC using SHA-256 in Java

```
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import org.springframework.security.crypto.codec.Hex;

public class CryptoUtils {

  public static String hmacSha256(byte[] input, String key) {
    try {
      Mac hmac = Mac.getInstance("HmacSHA256");   #A

      var hmacKey = new SecretKeySpec(key.getBytes(), "HmacSHA256");   #B
      hmac.init(hmacKey);                                              #B

      byte[] hash = hmac.doFinal(input);    #C

      return String.valueOf(Hex.encode(hash));    #D
    } catch (NoSuchAlgorithmException | InvalidKeyException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A get an implementation of the HMAC based on the SHA-256 algorthim.
#B initialize the HMAC with a secret key
#C compute the HMAC value on the input
#D return the HAMC value as a Hex String

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"[26] contains a list of all the algorithm names that can be passed to `Mac.getInstance()` method. The `SecretKeySpec` is the generic class for storing the bytes of a key along with the algorithm type that the key is supposed to be used this. The generic nature of `SecretKeySpec` makes it possible to use it 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.

> **TIP** The Java standard APIs for cryptography are low level and are easy to misuse. Therefore, you might want to use a higher-level library that simplifies and implements best practices for working Java cryptography APIs. Chapter 7 will show you how to use Google Tink[27] a developer friendly crypto library for computing an HMAC.

[26] https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html
[27] https://github.com/google/tink

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 for what HMAC is and how it is used will make understanding security documentations significantly easier.

3.5.2 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 `refunds.json` was created by the warehouse application and that it was not tampered with after it was created.

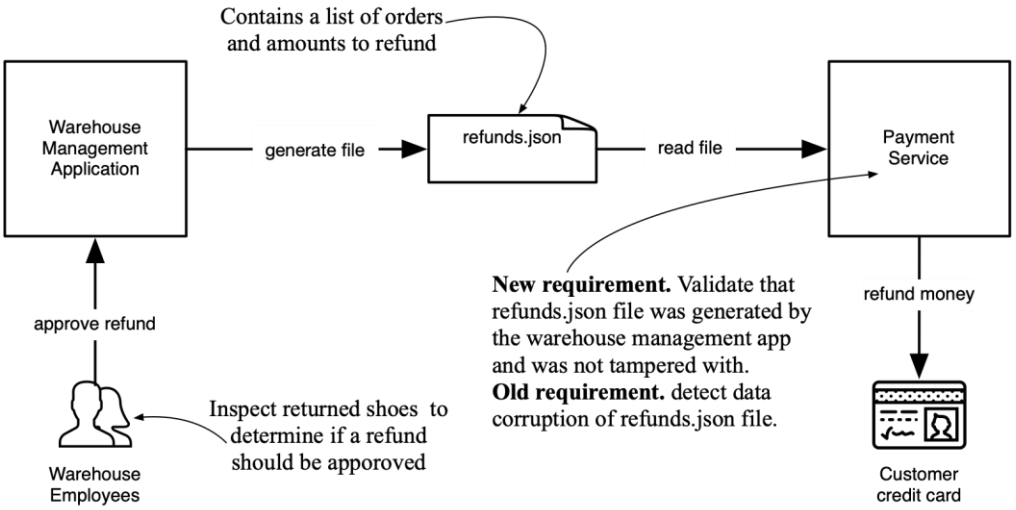


Figure 3.10 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service issues refunds to customer credit cards for the amount specified in the refunds.json file.

Listing 3.9 Example refunds.json file produced by the warehouse application

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

When the payment service retrieves the `refunds.json` file it computes the `HMAC-SHA256(refunds.json, secret key)` 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 to 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. The following diagram explains the validation flow visually.

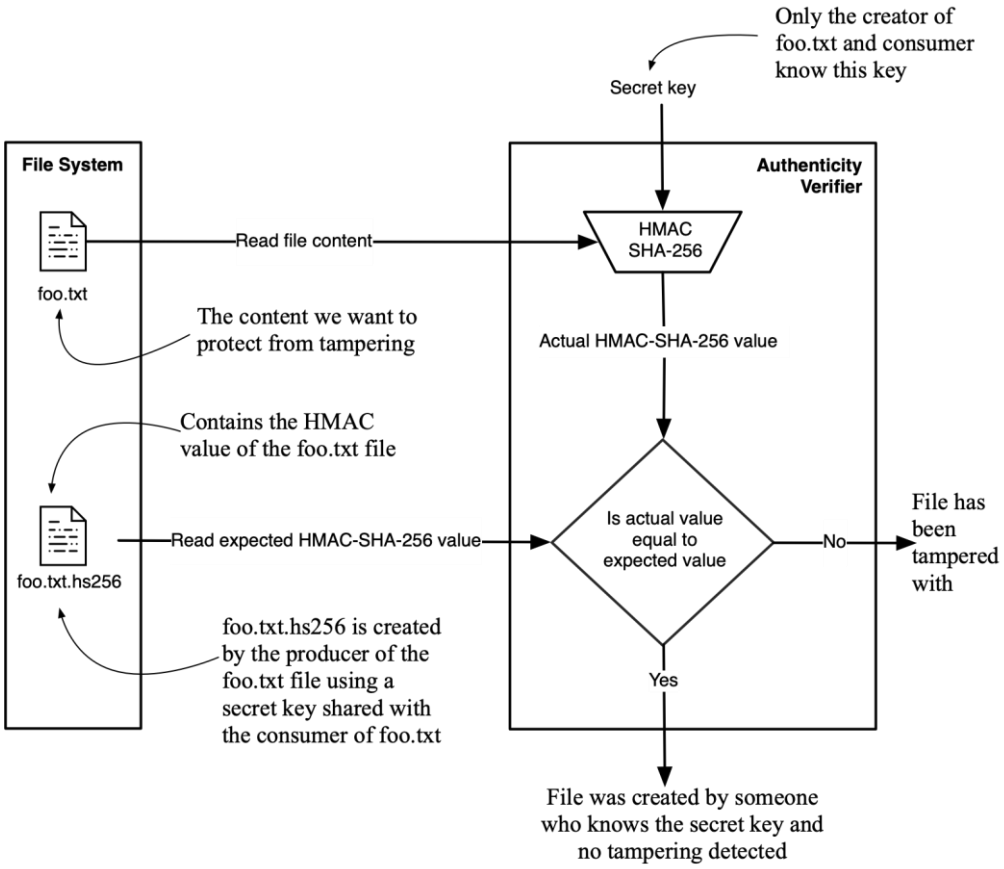


Figure 3.11 Determine who created a file and that the file was not tampered with using a Hashed Message Authentication Code (HMAC). The producer of the foo.txt file computes the HMAC-SHA-256 value of the content and writes it to foo.txt.hs256. A consumer computes the SHA-256 value of the contents of 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.

SAMPLE CODE The crypto-hmac sample application located https://github.com/cloud-native-security-patterns/crypto-hmac implements the above pattern in Java using HMAC with SHA-256 hash function. The code in the linked git repo provides detailed instructions for how to run the sample applications. I highly recommend run the sample application to deepen your understanding of the topic.

An HMAC requires a secret key to be shared between the warehouse application and the payments service. How does the warehouse application and the payment service agree on the on the value of this secret key? In this chapter we make the simplifying 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. The payment and warehouse applications store the secret key in the standard Spring Boot application.yml configuration file.

Listing 3.10 application.yml used to store the HMAC secret key

```
refunds:
  key: "262878bf9f89cbc3d7912a26cc272e6c"
```

WARNING Storing a key in configuration file in clear text is a security anti-pattern. The book has a part dedicated to securely storing secrets in credential storage services. Until we get to credential handling section of the book, we will use the Spring Boot configuration file to store secrets to minimize complexity and make the samples easier to understand.

The following code listing shows how the warehouse application generates the `refunds.json` and `refunds.json.hs256` files.

Listing 3.11 application.yml used to store the HMAC secret key

```
public class RefundGenerationService {

  public void generateReport(Path refundsFile, List<Refund> refunds, String key) {
    try {
      String refundsJson = JsonUtils.toJson(refunds);     #A
      Files.writeString(refundsFile, refundsJson);        #A
      String hashValue = CryptoUtils.hmacSha256(refundsJson.getBytes(), key);   #B

      String hmacFileName = refundsFile.getFileName() + ".hs256";   #C
      Path hashFile = refundsFile.resolveSibling(hmacFileName);     #C
      Files.writeString(hashFile, hashValue);                       #C
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A generate the refunds.json file
#B compute HMAC based on SHA-256 and the secret key
#C write computed HMAC to .hs256 file

The HMAC is `7387029822ad68e99326e1c14c7362e53cb3a4f4fcafa010e3590e9c832e529a` for the following JSON

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

By computing the HMAC-SHA256 hash of the refunds file and comparing it with expected hash stored in the `refunds.json.hs256` file payment service is able to detect data corruption and ensure that the file was generated by someone who has the secret key used to generate the message authentication code.

```
public class PaymentService {

  public void processRefunds(Path refundsFile, String key) {
    try {
      byte[] refunds = Files.readAllBytes(refundsFile);            #A
      String actualHmac = CryptoUtils.hmacSha256(refunds, key);    #A

      String hmacFileName = refundsFile.getFileName() + ".hs256";   #B
      Path hashFile = refundsFile.resolveSibling(hmacFileName);     #B
      String exceptedHmac = Files.readString(hashFile);             #B

      if (!actualHmac.equals(exceptedHmac)) {      #C
        throw new CorruptRefundFileException();    #C
      }

      System.out.println("Issuing Refund to");
      System.out.println(Files.readString(refundsFile));
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A compute the actual HMAC value
#B determined the expected HMAC value
#C check that the HMAC values match

The rest of the payment service and the warehouse application is identical to the cryptographic hash function example form the last section, so we are not repeating the supporting code in this section. Full code is available in the sample repo[28].

HMAC is a widely used by numerous security protocols such as the Transport Layer Security (TLS). The understanding of HMAC you have gained in this chapter will help you understand numerous security protocols and libraries. The table below shows which goals of cryptography HMAC delivers on.

Table 3.4 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.

[28] https://github.com/cloud-native-security-patterns/crypto-hmac

3.5.3 Exercises

1. What are the similarities and differences between a MAC and an HMAC?
2. What is the difference between HMAC-SHA-256 and HMAC-SHA-3-512?
3. Should you write your own cryptographic hash function?
4. What security guarantees does an HMAC function provide?

3.6 Summary

- Integrity, authentication, confidentiality, and non-repudiation are the four goals of cryptography.
- Java Cryptography Architecture (JCA) and Java Cryptography Extensions (JCE) provide extensive support for cryptography in Java. However, JCE and JCA are low level APIs that are easy to misconfigure and misuse.
- Use a developer friendly cryptography library that is hard to misuse such as Google Tink in production applications. We will cover Google Tink in chapter 6.
- Cryptographic hash functions such as SHA-2 and SHA-3 can be used to guarantee integrity only.
- Message authentication codes such as HMAC are used to guarantee integrity and authenticity of data. HMAC is built on using a cryptographic hash function.
- Cryptographic hash functions are always under attack by cryptographers looking for weakness in the algorithms. Before writing production code, always check the latest practices on which algorithms are still considered secure, and what the minimum key lengths should be. You can use keylength.com for key size recommendations.
- Consult with your corporate information security team for guidance on current recommend algorithms and key sizes.
- The examples in this book are optimized for educational value, they take shortcuts such as ignoring proper error handling, to make the code fit on the page. Use the samples to learn the security concepts. Don’t copy and paste the sample code blindly, you must make it production ready before you use it.

4

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 chapters 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.

4.1 ACME Inc. Scenario

Recall the ACME Inc. online shore 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.

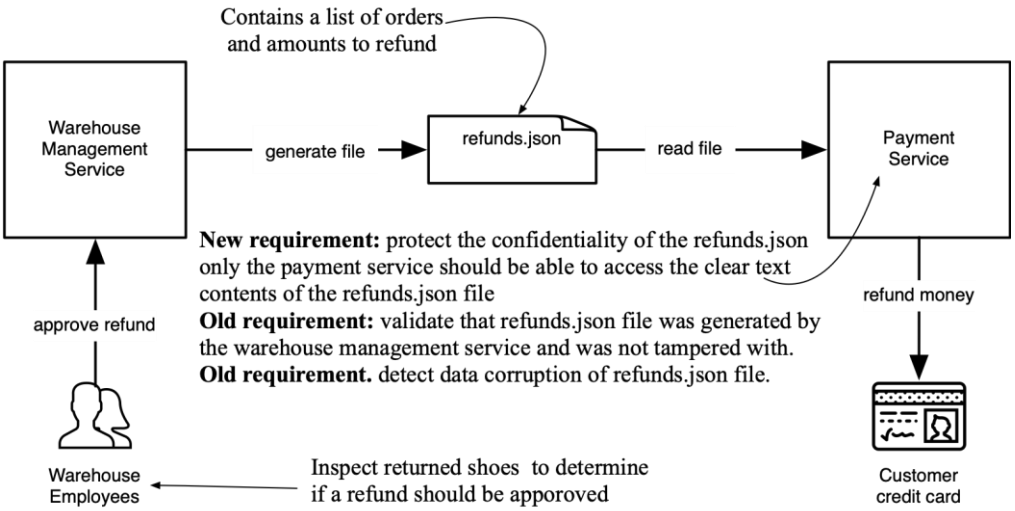


Figure 4.1 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service refunds customer credit cards for the amount specified in the refunds.json file.

Listing 3.1 Example refunds.json file produced by the warehouse application

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

In the last chapter we used a *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. By the end of the chapter, you will learn how to use the *Advanced Encryption standard* (AES) in *Galois Counter Mode* (GCM) to provide data integrity, authenticity, and confidentiality. You will also learn a lot of foundational concepts about encryption. Grab a coffee there is a lot of practical foundational security knowledge coming your way.

4.2 Advanced Encryption Standard Overview

To ensure confidentiality of data in transit or storage the data must be encrypted. Encrypting data scrambles, it is using a key, so the encrypted data is indistinguishable from a random sequence of bits. Decryption reverses the scrambled data back into the original data. The data being encrypted is referred to as the plain text and the encrypted data is called the cipher text.

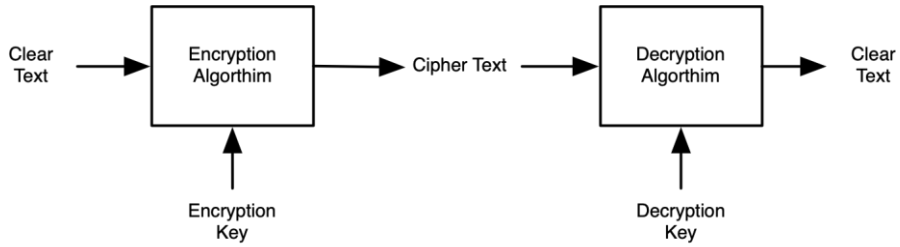


Figure 4.2 A cipher consists of an algorithm for encrypting and decrypting data using a key. The details of encryption and decryption algorithm are public knowledge while the key is kept secret. If the same key is used to encrypt and decrypt the cipher is called a symmetric cipher otherwise it is called an asymmetric cipher.

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 super computers is a huge engineering effort. The Advanced Encryption (AES) standard is very popular encryption algorithm, it is the de-facto standard for symmetric key encryption algorithm used in all applications including cloud native applications. All major CPU architectures —Intel, ARM and others— provide hardware support and acceleration for AES, 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*. Since AES uses the same key to encrypt and decrypt data it is classified as a symmetric block cipher. AES encryption keys can be 128, 192, 256 bits.

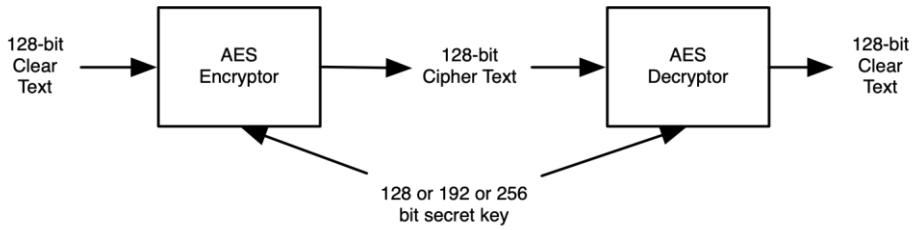


Figure 4.3 AES is a block cipher because it requires its input be a fixed size 128-bits long. AES is a symmetric cipher because the key used to encrypt is the key used to decrypt.

WARNING The AES standard defines many 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 do 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 secure at the time of writing. Corporate security teams publish recommendations for AES configurations that developers should use when working with AES. Consult with your information security team for recommended AES configurations.

4.3 Mode of operation

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

Suppose we want to encrypt a 1605-byte message with AES. Since AES only works on 16-byte blocks we will need to 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. For example, in the *Electronic Code Book* (ECB) mode of operation, each 16-byte block is encrypted independently using the secret key then the cipher text is concatenated to produce the encrypted output.

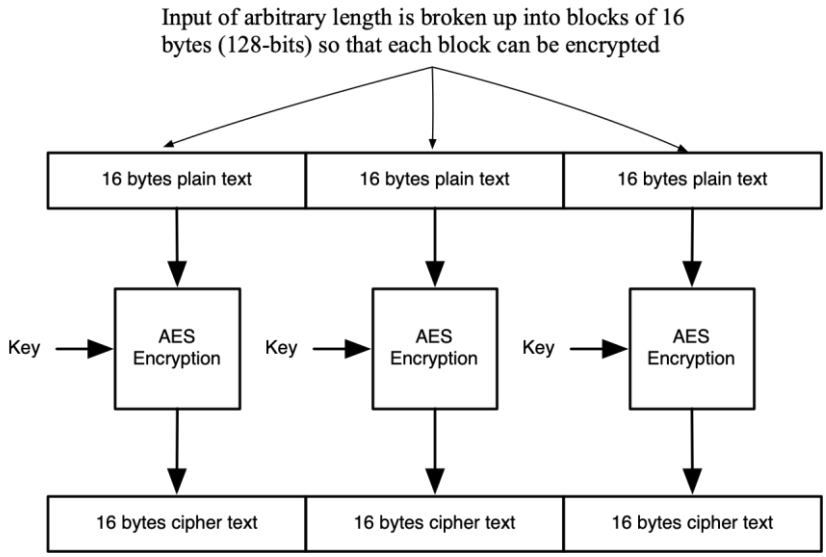


Figure 4.4 The Electronic Codebook Operating (ECB) Mode encrypt each input block independently using the secret key, concatenates the resulting cipher text. ECB mode is simple, but it is insecure, never use ECB mode.

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 how to implement 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 accidently.** 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 size input blocks. There are many modes of operations to choose from. AES supports the following modes of operation:

- *Electronic Code Book* (ECB)
- *Cipher Block Chaining* (CBC)
- *Cipher Feedback (CFB)*
- Output Feedback Mode (OFB)
- Counter Mode (CTR)
- *Galois Counter Mode* (GCM)
- *Synthetic Initialization Vector* (SIV)
- AES-GCM-SIV

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 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 is using?". Unless you know which mode is in use you cannot evaluate the security of the encrypted data.

AES supports a large number of operating modes because it is designed to be usable in a broad range of situations that demand different security, power consumption, and simplicity tradeoffs. For example, a low power temperature sensor installed in 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 should never be used. 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 not sure what AES mode to go 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.

4.3.1 Cipher Block Chaining (CBC) mode

Cipher Block Chaining (CBC) is an AES operating mode designed to ensure that every cipher text block is random even same plain text is encrypted twice. This means that an attacker cannot determine any patterns in the underlying data by looking for repeating patterns in the cipher text. To use AES in cipher block chaining mode you will need two things, a secret key and an *Initialization Vector* (IV).

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 a college mathematics

rest assured in the context of AES initialization vector is a 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 below provides a visual explanation of CBC operating mode.

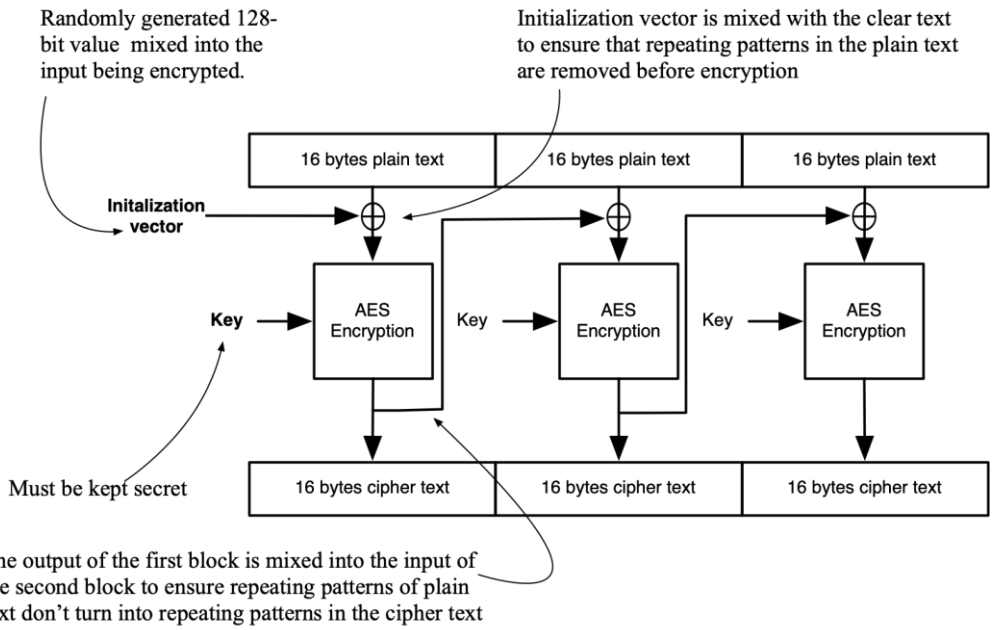


Figure 4.5 The symbol with + inside a circle is the XOR operation. Cipher Block Chaining (CBC) takes the output of one block and feeds it into the next block. The net result is that even if you encrypt the same data multiple times the cipher text looks random. An attacker cannot determine any patterns in the underlying data by looking for patterns in the cipher text.

The initialization vector 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 (IV) 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. 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, to produce an output consisting of the initialization vector, followed by the cipher text. The diagram below provides a visualization of the AES in CBC mode mental model.

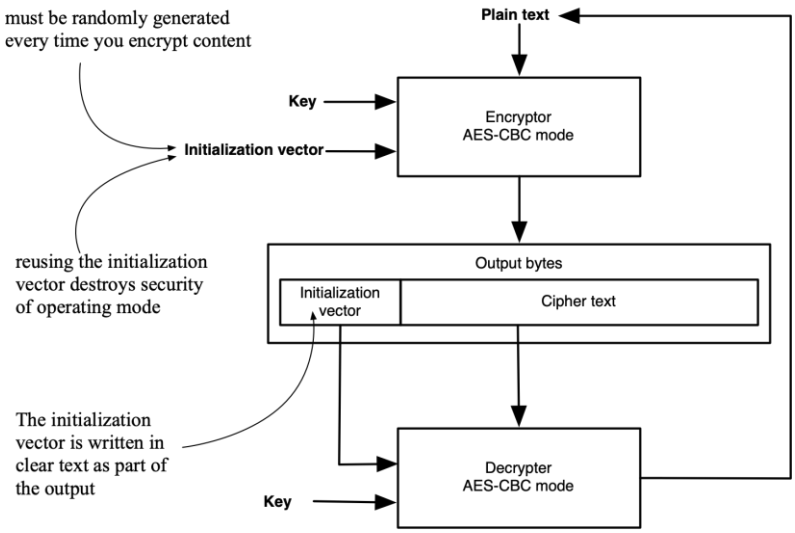


Figure 4.6 AES in CBC mode can be visualized as a black box that takes three inputs: plain text, key, and initialization vector. The initialization vector is written out as a plain text to output followed by the cipher text computed by the AES in CBC mode. Decryption process requires three inputs: initialization 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.

> **WARNING** If a key and initialization vector combination is reused you can end up with catastrophic failure of the encryption. 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 impact. Key take away is that you should never ever 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`.

Suppose that a file is encrypted with AES-CBC, an attacker can tamper with the encrypted file 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[29] 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 confidentiality of the encrypted data it does not protect integrity or the authenticity of the data. The table below shows security goals can be realized by AES-CBC operating mode.

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

Table 4.1 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.

4.3.2 Authenticated encryption

As we saw in the previous section, data confidentiality by itself is not enough; an attacker can tamper with encrypted data in highly determinantal ways to consuming applications. Authenticated encryption refers to any encryption scheme that provides data integrity, authenticity, and confidentially.

Authenticated encryption can be implemented by combining message authentication codes with an encryption algorithm. For example, data is encrypted with AES-CBC mode 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 combine encryption and Message Authentication Codes (MAC).

- **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 compute the MAC on the plain text then encrypt both the plaintext and the mac, this approach is used by the Transport Layer Security (TLS) 1.2 protocol. Mac-then-Encrypt should only be used by cryptography experts 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 message authentication codes correctly is tricky. Cryptographers designed several modes of operation for AES that provide authenticated encryption. Galois Counter Mode (GCM) is the one of the mostly widely used authentication encryption mode and is the subject of the next section.

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

4.3.3 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 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.

Think of AES-GCM mode 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. The diagram below provides a visualization of the AES in GCM.

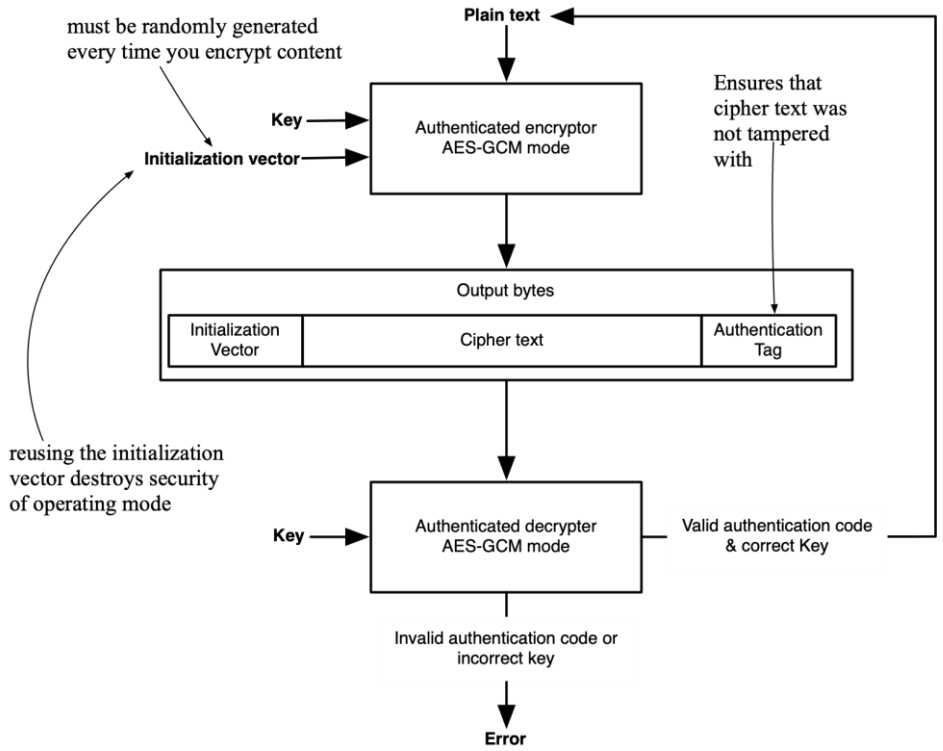


Figure 4.7 AES in GCM mode can be visualized as a black box that takes three inputs: plain text, key, and initialization vector. The initialization vector is written out as a plain text to output followed the cipher text computed by the AES in GCM mode algorithm, followed by the authentication code computed by GCM mode. Decryption process requires four inputs: initialization vector, cipher text, and key, and authentication code.

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 4.2 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 super easy to make mistakes when writing code that uses AES for example re-using an initialization vector, or not using enough randomness, or any of other ways of incorrectly coding. Secure coding practices and code reviews are essential ensuring you don't accidently introduce a vulnerability.

TIP use a developer friendly crypto library that takes care of configuring commonly used encryption algorithms correctly. Google Tink is an excellent choice for a developer friendly library which we will cover in an upcoming chapter https://github.com/google/tink

4.4 Java Support for AES

Java 11 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 the listing below generates the requested number of random bytes using a cryptographically secure random number generator.

Listing 4.2 Generating Secure Random Numbers

```
import java.security.SecureRandom;

public class CryptoUtils {

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

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

#A always use a secure source of randomness when generating initialization vectors for AES
#B securely generate the specified number of random bytes

WARNING 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 accidentally import `Random` instead of `SecureRandom`.

The code listings below show a utility function that take an array of bytes and encrypts using AES in GCM operating mode with a 256-bit key.

Listing 4.3 Encrypting with AES/GCM

```
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import org.springframework.security.crypto.util.EncodingUtils;

public class CryptoUtils {

  public static byte[] encryptAes256Gcm(byte[] clearText, byte[] key) {
    try {
      Cipher aes = Cipher.getInstance("AES/GCM/NoPadding"); #A

      var gcmParameterSpec = new GCMParameterSpec(128, randomBytes(12)); #B
      var aesKey = new SecretKeySpec(key, "AES");
      aes.init(Cipher.ENCRYPT_MODE, aesKey, gcmParameterSpec);

      byte[] cipherText = aes.doFinal(clearText); #C

      return EncodingUtils.concatenate(gcmParameterSpec.getIV(), cipherText); #D
    } catch (NoSuchAlgorithmException
        | NoSuchPaddingException
        | InvalidKeyException
        | InvalidAlgorithmParameterException
        | IllegalBlockSizeException
        | BadPaddingException e) {
      throw new RuntimeException(e);
    }
  }
```

#A get an implementation of the AES algorithm in GCM mode with no padding
#B Configure the cipher to encrypt and generate 128-bit authentication tags and to use a 12-byte initialization vector
#C Encrypt the plain text
#D add the initialization vector just before the cipher text

The listing above 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"[30] 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 the all the data to be encrypted with using the `doFinal()` method alternatively you can call the `update()` method and pass the data in chunks which can be useful in you 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 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.

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

Listing 4.4 Decryption

```
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import org.springframework.security.crypto.util.EncodingUtils;

public class CryptoUtils {

public static byte[] decryptAes256Gcm(byte[] input, byte[] key) {
    try {
      Cipher aes = Cipher.getInstance("AES/GCM/NoPadding");

      byte[] iv = EncodingUtils.subArray(input, 0, 12); #A
      byte[] cipherText = EncodingUtils.subArray(input, 12, input.length); #A

      var gcmParameterSpec = new GCMParameterSpec(128, iv); #B
      var aesKey = new SecretKeySpec(key, "AES"); #B
      aes.init(Cipher.DECRYPT_MODE, aesKey, gcmParameterSpec); #B

      return aes.doFinal(cipherText); #C
    } catch (NoSuchAlgorithmException
        | NoSuchPaddingException
        | InvalidKeyException
        | InvalidAlgorithmParameterException
        | IllegalBlockSizeException
        | BadPaddingException e) {
      throw new RuntimeException(e);
    }
  }}
```

#A separate the input into the initialization vector and the cipher text
#B Configure the cipher
#C decrypt the input

> **TIP** The Java crypto APIs are low level it is easy to accidentally misuse them and introduce vulnerabilities into your application. Consider using a higher-level library that simplifies and implements best practices for working with the Java cryptography libraries. 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.

4.4.1 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.

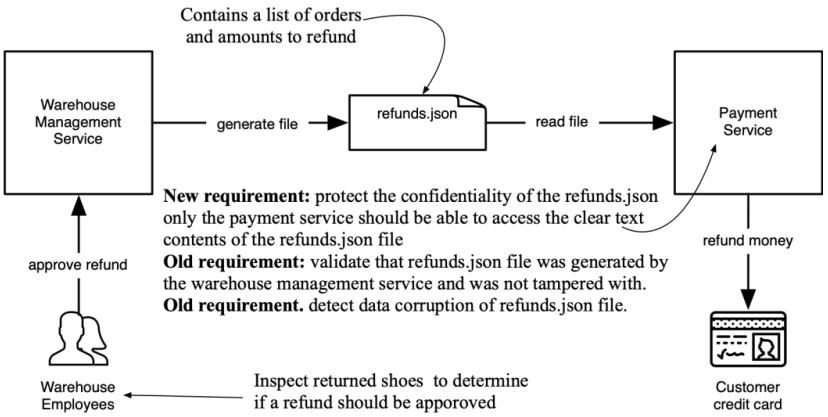


Figure 4.8 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service refunds customer credit cards for the amount specified in the refunds.json file.

To guarantee integrity, authenticity and confidentiality of the `refunds.json` file the warehouse service encrypts it using AES-GCM mode with a 256-bit key. The payments service 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 `refunds.json` was created by the warehouse service.

> **SAMPLE CODE** The crypto-aes sample application located https://github.com/securing-cloud-applications/crypto-aes implements the above pattern in Java using AES/GCM 256-bit key. The code in the linked git repo provides detailed instructions for how to run the sample applications. I highly recommend run the sample application to deepen your understanding of the topic.

Listing 4.5 Generating the refund

```
public class RefundGenerationService {

  public void generateReport(Path refundsFile, List<Refund> refunds, String key) {
    try {
      String refundsJson = JsonUtils.toJson(refunds);
      byte[] cipherText = CryptoUtils.encryptAes256Gcm(         #A
                                    refundsJson.getBytes(),    #A
                                key.getBytes());          #A
      Files.write(refundsFile, cipherText);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A encrypt the refund using the utility function from listing 4.3

A single a call to `CryptoUtils.encryptAes256Gcm()` utility function is all we need to generate an encrypted and authenticated file. The code is simpler than the HMAC example while providing all the benefits of HMAC in addition to data confidentiality. Decryption is equally as simple and is illustrated blow.

Listing 4.6 Generating the refund

```
public class PaymentService {
  public void processRefunds(Path refundsFile, String key) {
    try {
      byte[] clearText= CryptoUtils.decryptAes256Gcm(                #A
                                    Files.readAllBytes(refundsFile), #A
                                     key.getBytes());                 #A
      String refundsJson = new String(clearText);
      System.out.println("Issuing Refund to");
      System.out.println(refundsJson);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A encrypt the refund using the utility function from listing 4.4

In the example code above using AES in GCM what do you expect to happen if the encrypted file is corrupt? If CBC operating mode was used instead of GCM and the refunds file was corrupted will `processRefunds()` method in the listing above detect the fie corruption? If the file is corrupt AES in GCM mode will raise an error, in CBC mode the decrypted file will be corrupt.

4.5 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 is used by intermediary systems to route the message to its final destination. The message body contains content that must be kept confidential. Therefore, the message body is encrypted. 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.

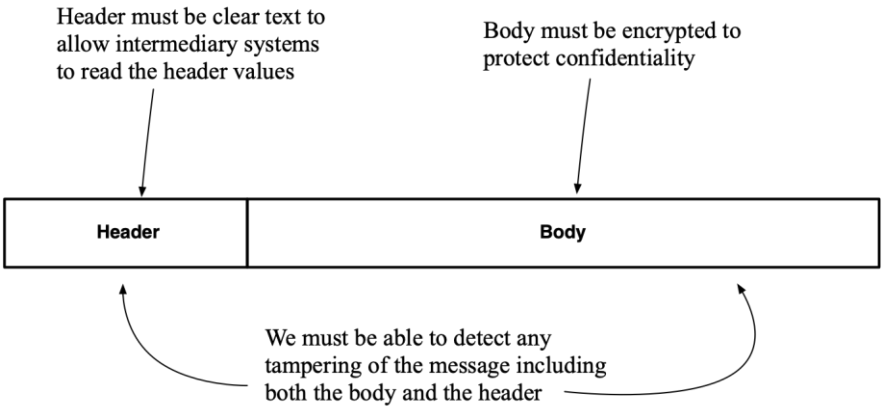


Figure 4.9 A message with a plain text header and an encrypted body is very common.

An attacker can intercept the message and modify the header causing the message to be routed to the wrong recipient. The attacker can intercept two messages then swap the message bodies causing problems for the receiving application. Therefore, the receiving application must be able to detect if the header has been tampered with, or if the encrypted body has been swapped. In security terminology the header is referred to as Additional Associated Data (AAD) because it is plain text that is related and bound to a specific cipher text.

We can use a message authentication code computed over the header and the body to defend against tampering of the body or the header. This approach requires the use of two algorithm HMAC and AES. There is an easier way that requires only a single algorithm.

As you learned in the previous section an authenticated encryption mode does the combined work of encrypting and creating a message authentication code. Authenticated Encryption with Associated Data (AEAD) refers to operating modes that can keep some of the data in plain text, encrypt the rest and compute a MAC of the undecrypted and encrypted components. It does the work of HMAC(header || AES(Body)) in a single algorithm. AES in GCM mode supports AEAD which accounts for the popularity of the AES GCM mode.

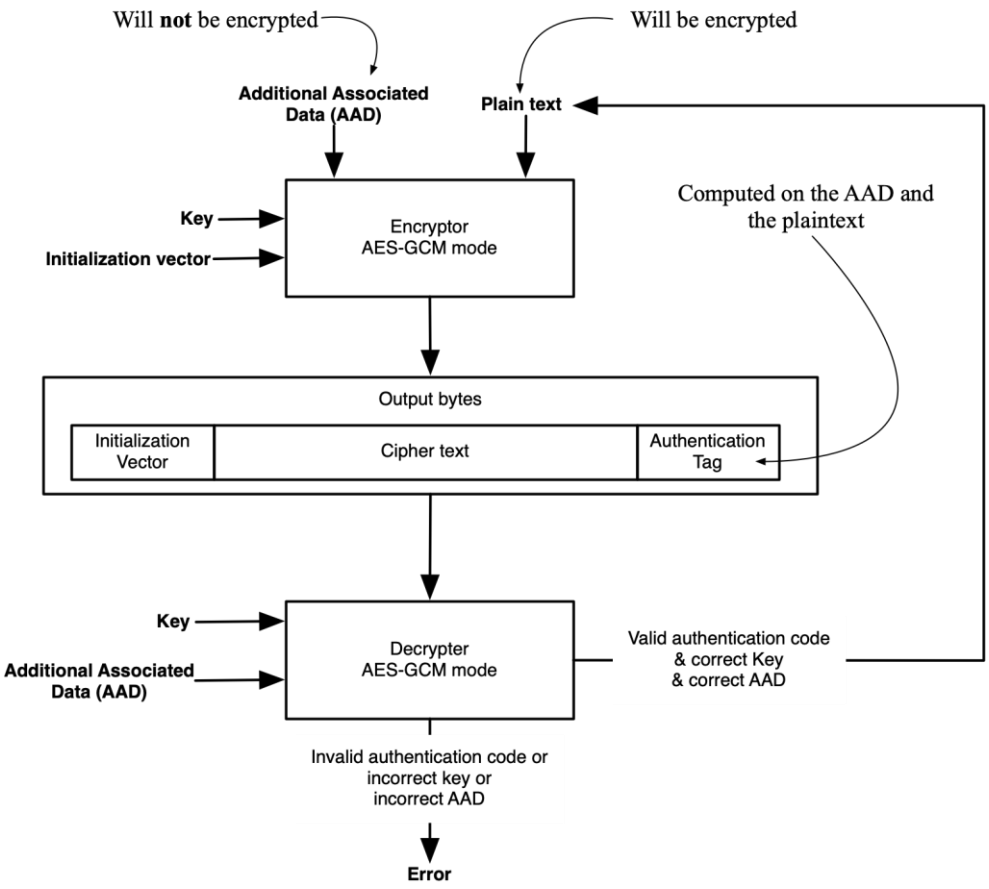


Figure 4.10 AES in GCM mode can be visualized as a black box that takes four inputs: plain text, secret key, initialization vector, and optional Additional Associated Data (AAD). The authentication tag is computed using the AAD data and the cipher text. The initialization vector, cipher text, and authentication 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 or hacker or 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 with has

been tampered with. Some databases such as Google Big Query have native support for AEAD[31].

4.6 Data encryption and compression

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 then encrypted, or should we encrypt the data first 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 stream of data without any repeating patterns. Therefore, compression algorithms perform poorly against encrypted cipher text. It is better to compress data first, then encrypt it so that you get the maximum compression ratios.

4.7 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 256-bit key, below more details about this recommendation.

4.7.1 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.

To brute force a 128-bit AES key using an electronic computer requires quadrillions of years of compute 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 if what if you have a quantum computer?

Quantum computers are 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 256-bit key the strength of 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 implementation of AES. An AES implementation may contain bugs which 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

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

cases of developers publishing secret keys to GitHub[32] 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.

4.7.2 Checklist for using AES-GCM correctly in Java

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

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 (239 – 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 accidently 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.

4.8 The problem with shared secrets

The AES algorithm assumes that the producer and consumer of encrypted data share the AES secret key. Sharing a secret key between the various systems that need to use the key

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

becomes an error prone challenging automation problem. In a chapter 6 "Public Key Encryption and Digital Signatures" we will solve the secret sharing problem.

4.9 Exercises

Computer security has a lot of jargon and acronyms that get used as words. It can be quite confusing to read a sentence that has 4 security acronyms in it and to figure out what it all means. A good way to get familiar with acronyms is to use them in a sentence so take a few minutes and write down an answer to following questions to get the acronyms and concepts in this chapter firmly into your mind.

1. What is an AES mode of operation?
2. What is authenticated encryption?
3. What is the difference between authenticated encryption and authenticated encryption with additional data?
4. What do these acronyms stand for AES, AE, AEAD?
5. What capabilities does the AES GCM operating mode provide?
6. Is the AES-CBC mode safe to use and why?

4.10 Summary

- At the time of writing the Advanced Encryption Standard (AES) is a widely deployed highly secure symmetric block cipher with numerous software and hardware implementations.
- AES can only encrypt 128-bits at a time it must be combined with an operating mode to encrypt more than 128 bits of data. Some operating modes are insecure and should not be used.
- Authenticated encryption refers to AES modes that provide data integrity, authenticity, and encryption.
- Always use an AES authenticated encryption mode such as Galois Counter Mode (GCM).
- Additional Associated Data (AAD) refers to plain text data that is used in computing the authentication tag produced by an authenticated encryption mode.
- Authenticated Encryption with Associated Data (AEAD) refers to encryption operating modes that support encrypting plain text and using additional associated data to compute the authentication tag.
- AES-GCM is an authenticated encryption with associated data operating mode which accounts for its popularity.
- Use a developer friendly cryptography library that is hard to misuse such as Google Tink in production applications.
- 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 emphasis the concepts. Don't copy and paste the sample code blindly, you must make it production ready before you use it.

5

JSON Object Signing and Encryption (JOSE)

This chapter covers

- Identify the component that make 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. Standardized data formats for exchanging encrypted and signed data makes interoperability significantly easier.

Security protocols such as X.509 digital certificates, OpenID connect, OAuth2, SAML, TLS 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 will be able to 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. Debugging and troubleshooting is simple when you understand the underlying data formats and hard 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. Chapter 17 covers JWT in depth in the context of the OpenID Connect standard.

5.1 The Standards Layer Cake

The JavaScript Object Signing and Encryption (JOSE) is a collection of four standards that build on each other. 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 3 is identified using the string HS256.
2. *JSON Web Key (*https://tools.ietf.org/html/rfc7517) (JWK) is used to represent 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) is used to represent 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) is used to represent content that must be protected from tampering and must be kept confidential. The JWS standard uses the JSON Web Algorithms (JWA) and JSON Web Key (JWK) standard.

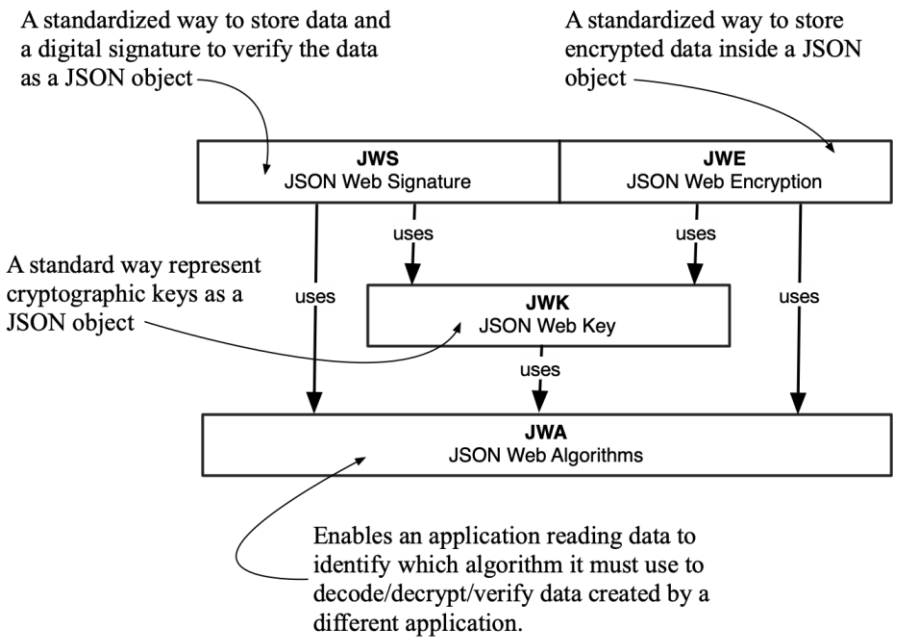


Figure 5.1 Relationship between the standards that are part of the JSON Object Singing and Encryption (JOSE) suite. JOSE enables interoperability between applications that want to exchange encrypted or signed data using the JSON data format. JOSE standards is used extensively by popular security protocols such as OpenID Connect for Single Sign On.

5.2 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 that 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 applications are written by different teams working for different companies, 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. Another team might choose to use the string "AES-GCM-128" to indicate that the encryption algorithm is Advanced Encryption Standard with 128 bits key 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, message authentication codes, key management, and encryption. Some example 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 adding 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, and root DNS domains, list of media content contents. You can see the full list of JWA names at https://www.iana.org/assignments/jose/jose.xhtml. As developer you will be able to use the JWA when you are debugging to identify what type of algorithms are being used.

5.3 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 pair, 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 3 we stored HMAC keys as hex encoded string in the Spring Boot `application.yml` file. This approach works when you have two applications written by the same team, but it does scale for multiple teams spread across multiple organizations. The *JSON Web Key* (JWK) is a standard for representing keys as JSON objects. JWK is helpful for representing complex keys that have multiple components, for example, the JSON below represent an elliptic curve public private key pair using the NIST P-256 curve. We will discuss ecliptic curve cryptography in the next chapter.

Listing 5.1 Example JSON Web Key

```
{
  "kty":"EC",
  "crv":"P-256",
  "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
  "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
  "d":"870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
  "use":"enc",
  "kid":"1"
}
```

We will cover various types of JWK keys in the sections where we discuss the algorithms that the keys are based on. The chapters on public key encryption, and OpenID connect will discuss JWK in the context where it is used.

5.4 JSON Web Signature (JWS)

Recall the ACME Inc. online shoe retailer refund processing scenario we used in the previous chapter. 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 a `refunds.json` file is produced by the warehouse management application and sent to the payments service to refund customer credit cards.

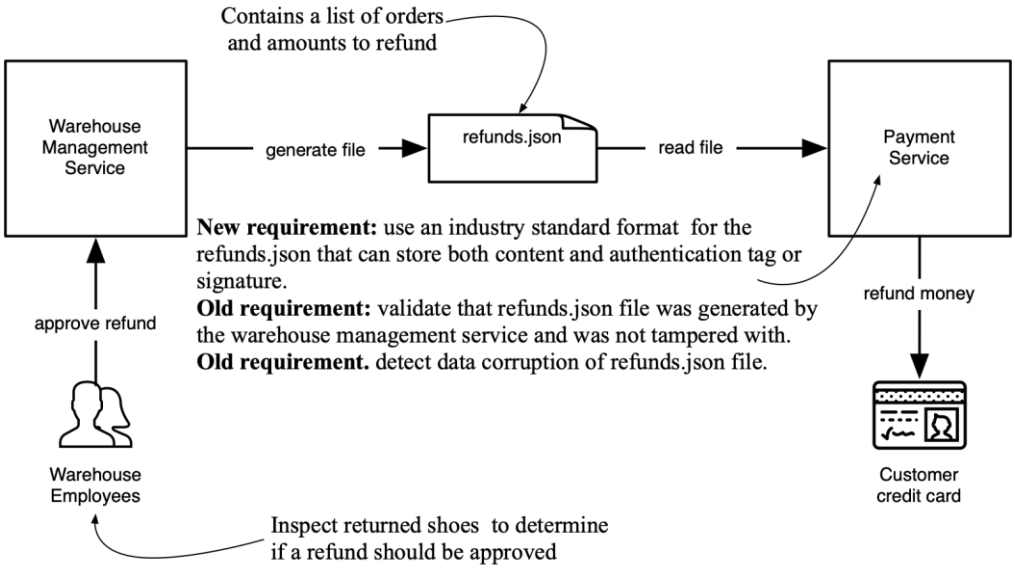


Figure 5.2 ACME Inc. staff approve refunds using the warehouse management application. Once a day the warehouse management application generates a refunds.json file. The payment service refunds customer credit cards for the amount specified in the refunds.json file.

In chapter 3, we used HMAC with SHA-256 to authenticate that the `refunds.json` file was generated by the warehouse service and that it was not tampered with. However, the implementation in generates two files: `refunds.json.hs256` and `refunds.json`.

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. 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 used by the OpenID connect

standard and it can be used for a variety of microservice security applications. We can use it to generate a single file that contains both the data and the message authentication code. JWS is part of a collection of IETF standards called JavaScript Object Signing and Encryption (JOSE).

5.4.1 JSON Web Object (JWS) Structure

A JWS object has three parts: header, body, and signature. 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.

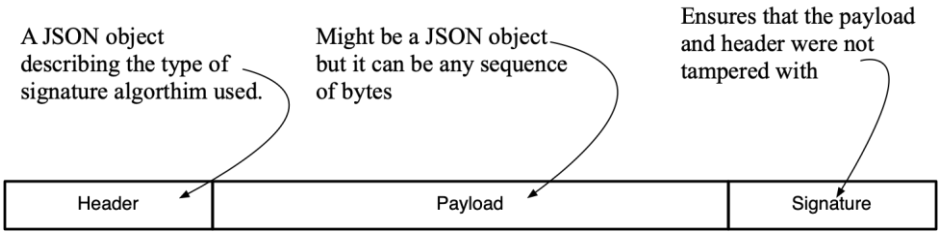


Figure 5.3 The logical structure of a JSON Web Signature (JWS). A JSON metadata header describing the type of signature used. A payload that can be any type of data format not just JSON. Signature to use verify that the header and payload were not tampered with. A JWS payload is always readable to anyone who can access the JWS object.

For example, the JSON payload in listing 5.2

Listing 5.2 example payload that we want to sign convert into a JWS

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

turns into the JWS in listing 5.3 when it is signed using the HMAC with SHA-256. If you want a refresher on how Hashed Message Authentication Codes (HMAC) work review chapter 3.

Listing 5.3 example payload that we want to sign convert into a JWS

```
eyJhbGciOiJIUzI1NiJ9.WyB7CiAgIm9yZGVySWQiIDogIjEyMzQ1IiwKICAiYW1vdW50IiA6IDUwMAp9LCB7CiAgIm
      9yZGVySWQiIDogIjU2Nzg5IiwKICAiYW1vdW50IiA6IDI1MAp9IF0.wuKUe-
      eeHzO7ekWMJNRC80LOQ9EB2ZqNDRQ-DQrpuWY
```

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 a HTTP header and query parameter values each part of the JWS object is encoded using Base64 URL encoding then concatenated together separated by dots.

> **TIP** Base 64 encoding is a scheme for encoding binary data in an ASCII string that uses alphanumeric characters (a...z, A...Z, 0...9), in addition to '=', '+' and '/'. The '/' character must be escaped '%2F' when it is used as a value in a URL. To avoid having to escape base64 string in URL parameters or HTTP headers a variant called Base64 URL encoding is used where the '+' is replaced by '-' and '/' by '_' so that all characters need no escaping. Based64 URL encoding is used extensively by the *JavaScript Object Signing and Encryption* (JOSE) group of standards which includes JSON web signature (JWS). You can use Base64 URL encoded in your own applications when you need to include a string in a URL or an http header without wanting to escape it.

Diagram 5.4 shows the process used to convert the example JSON payload in listing 5.2 into the JWS in listing 5.3.

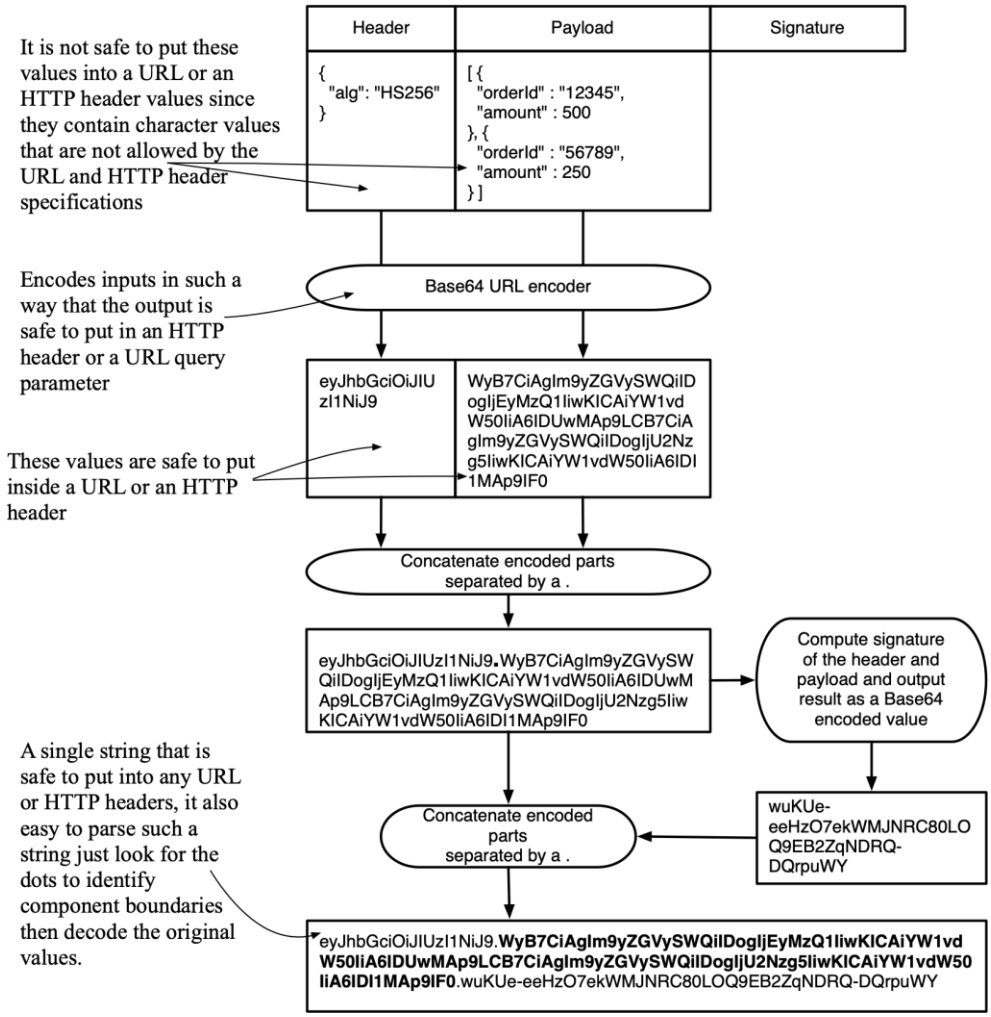


Figure 5.4 A JSON Web Signature (JWS) object can be safely embedded in a URL or HTTP headers because it is represented a base64 URL string where each component is separated by a dot.

The header of the JWS in figure 5.4 object is `eyJhbGciOiJIUzI1NiJ9` decodes to the JSON object

```
{
  "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[33]. `HS256` is the JSON web algorithm name for HMAC using SHA-256. The payload component of the JWS is

```
WyB7CiAgIm9yZGVySWQiIDogIjEyMzQ1IiwKICAiYW1vdW50IiA6IDUwMAp9LCB7CiAgIm9yZGVySWQiIDogIjU2Nzg
      5IiwKICAiYW1vdW50IiA6IDI1MAp9IF0
```

Which decodes to list of orders to refund produced by the warehouse application

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

The signature component `wuKUe-eeHzO7ekWMJNRC80LOQ9EB2ZqNDRQ-DQrpuWY` is the result of computing the HMAC-SHA-256 on the header and payload, it 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 or encryption, encoding, compression and data analysis. CyberChef is available at GitHub https://gchq.github.io/CyberChef/

A JWS object can be represented is 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 it not used with the various security protocols we discuss in the rest of the book. Depending on the signing the algorithm used a JWS can deliver on all the goals of cryptography. The signing algorithm used in this section does not support the non-repudiation cryptography goal, the next chapter on public key cryptography shows an example of using JWS to achieve the non-repudiation goal.

Table 5.1 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

[33] https://tools.ietf.org/html/rfc7518

5.4.2 Creating and verifying a JWS object

JSON Web Signature is widely supported with multiple implementations available in Java and other programming languages. Therefore, it is easy to create a JWS in a NodeJS application pass it to a Spring Boot Java based microservice that can easily validate it using one of the many Java libraries that support JWS. 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. Nimbus is used by Spring Security to implement support of OpenID Connect and OAuth2 login. The JWS object explained in the previous section was created using the code snippet below.

> **CODE SAMPLE** The crypto-hmac-jws sample application located at https://github.com/securing-cloud-applications/crypto-hmac-jws contains the sample code for creating and verifying JWS objects in Java. The code in the linked git repo provides detailed instructions for how to run the sample applications. I highly recommend running the sample application to deepen your understanding of the topic.

Listing 5.4 Create a JWS object and sign it using a secret key

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.JWSVerifier;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.MACSigner;
import com.nimbusds.jose.crypto.MACVerifier;
import java.text.ParseException;
import java.util.Optional;

public class CryptoUtils {

  public static String signJwsHmacSha256(String content, byte[] key) {
    try {
      JWSHeader header = new JWSHeader(JWSAlgorithm.HS256); #A
      Payload payload = new Payload(content);
      JWSObject jwsObject = new JWSObject(header, payload);
      jwsObject.sign(new MACSigner(key)); #B
      return jwsObject.serialize(); #C
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A Configure JWS to HMAC with SHA-256 for signing
#B Sign the JWS using a secret key and HMAC-SHA-256
#C Generate base64 URL encoded representation of JWS

To create a JWS signed with HS256 you will need some content to act as the payload and a secret to use for signing. Nimbus provides several classes for working with JWS.

- `JWSHeader` is used to represent JWS header in the above example we set to use HS256
- `Payload` is used to represent the payload of the JWS it can be any String
- `JWSObject` represents the combination of a header, payload and signature. The signature is created via the `sign()` method. In the above example listing we pass it a `MACSigner` configured with the secret key to use for computing the HMAC-SHA256.

When an application receives a JWS token it is important to verify the signature of the token to ensure that it has not been tampered with. Verifying the JWS token with the Nimbus library is easy as shown below.

Listing 5.5 Verify a JWS object using Nimbus

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.JWSVerifier;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.MACSigner;
import com.nimbusds.jose.crypto.MACVerifier;
import java.text.ParseException;
import java.util.Optional;

public class CryptoUtils {

  public static Optional<String> verifyJwsHmacSha256AndGetPayload(
                                          String jws, byte[] key) {
    try {
      JWSObject jwsObject = JWSObject.parse(jws);
      JWSVerifier verifier = new MACVerifier(key); #A
      if (jwsObject.verify(verifier)) { #B
        return Optional.of(jwsObject.getPayload().toString());
      }
      return Optional.empty();
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A configure the verifier
#B verify the JWS

The `JWSObject.parse()` method take a base64 url 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 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. 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 (https://www.zofrex.com/blog/2020/10/20/alg-none-jwt-nhs-contact-tracing-app/) 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 Nimubs MACVerifier we used in the sample listing 5.5 throws an exception if alg:none is used so the sample above is safe from alg:none vulnerability. Make sure to check that you are using your JWS/JWT library correctly.

5.4.3 Implementing credit card refunds scenario with JWS

To achieve all the goals stated at the start of this section we can use JWS to ensure integrity, authenticity using an industry standard data format that is easy to implement in Java.

Listing 5.6 Generate the refunds.jws file

```
public class RefundGenerationService {

  public void generateReport(Path refundsFile, List<Refund> refunds, byte[] key) {
    try {
      String refundsJson = JsonUtils.toJson(refunds);
      String jws = CryptoUtils.signJwsHmacSha256(refundsJson, key);
      Files.writeString(refundsFile, jws);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

The warehouse code to generate the refunds is radically simplified when we use JWS since we no longer have to deal with writing two files: refunds.json and refunds.json.sha25. Also, since JWS header provides metadata on what signing algorithm is used making it easy to switch algorithms or their configuration for example use SHA-512 instead of SHA-256. The payments services can easily read the refunds.jws, verify it came from the warehouse application, was not tampered with and extract the payload with the code it listing 5.7.

Listing 5.7 Verify refunds.jws and process refunds

```
public class PaymentService {

  public void processRefunds(Path refundsFile, byte[] key) {
    try {
      String jwe = Files.readString(refundsFile);
      String refundsJson =
          CryptoUtils.verifyJwsHmacSha256AndGetPayload(jwe, key) #A
              .orElseThrow(() -> new CorruptRefundFileException());
      System.out.println("Issuing Refund to");
      System.out.println(refundsJson);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A helper returns an java.util.Optional

The code to process a refund must be validate that the JWS object it is parsing has not been tampered with. Verification itself is delegated to the CryptoUtils class which in turn uses the Nimbus JOSE library to actually perform the validation and parsing of the JWS. Since the JWS might have been tampered with a `java.util.Optional<String>` is returned from the `CryptoUtils.verifyJwsHmacSha256AndGetPayload()`. An empty optional indicates that the JWS was not valid. If one key used to validate does not match the key used to sign the JWS the optional will be empty.

5.5 JSON Web Encryption (JWE)

In the previous section we used JSON Web signature to make sure that we can detect tampering in 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.

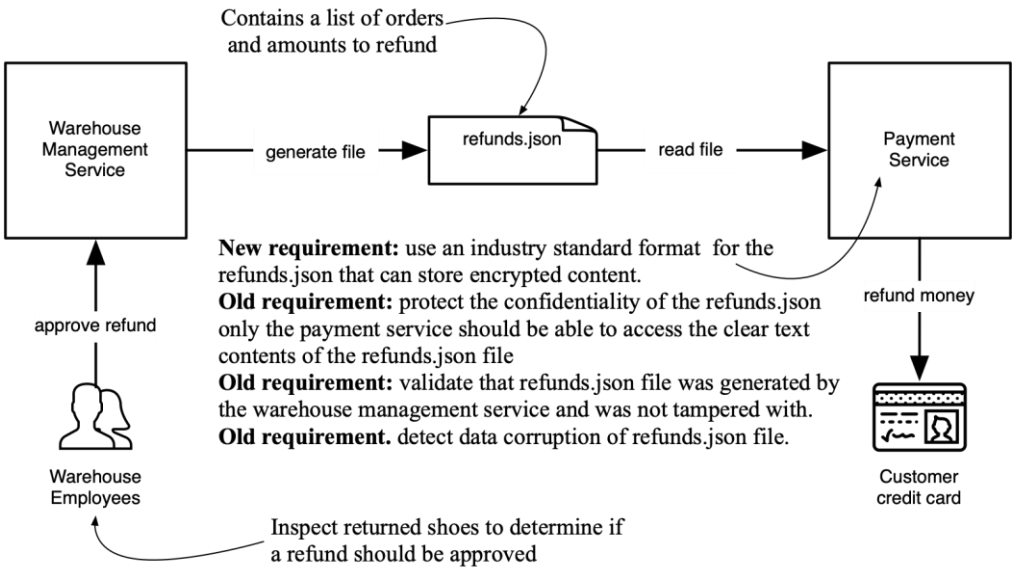


Figure 5.5 ACME Inc. staff inspect returned merchandise and approve refunds using the warehouse management service. Once a day the warehouse management service generates a refunds.json file and sends it to the payment service. The payment service issues refund to customer credit card accounts in the refunds.json file.

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."[34] (an octet is one byte).

> **TIP** this section assumes you are familiar with AES and symmetric key cryptography, review chapter 3 if you need a refresher.

5.5.1 JWE Structure

A JSON web encryption object has four logical parts: header, optional encrypted key, payload cipher text, and an authentication tag.

[34] https://tools.ietf.org/html/rfc7516

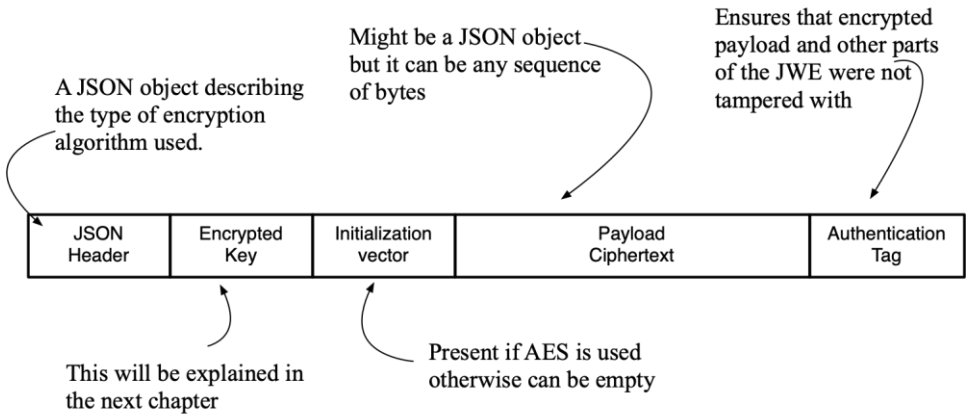


Figure 5.6 The logical parts of a JSON Web Encryption (JWE) object.

- *Header* is a JSON object that specifies which encryption algorithm is used and metadata for the algorithm configuration. JWE supports AES and other public key cryptography algorithms.
- *Encrypted key* is optional. It contains the encrypted value of the key that was used to encrypt the payload. Encrypting the encryption key is typically referred to as key wrapping and will be explained in the next chapter on public key cryptography.
- Initialization vector is optional because JWE can use algorithms that don't require an initialization vector. If the JWE is using AES, then an initialization vector it is included in the JWE object. The generation and management of the initialization vector is typically managed by the JWE library you are using. As a developer you should not have to manage the initialization vector yourself.
- *Payload ciphertext* is the encrypted content that is protected by the JWE it is an arbitrary sequence of bytes. It can be a JSON object, a string or anything else.
- *Authentication tag* is the message authentication code used to verify that the JWE object has not been tampered with.

For example, the JSON payload in listing 5.8

Listing 5.8 example payload that we want to encrypt into a JWE

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

turns into the JWE in listing 5.9 when it is encrypted using the AES-GCM using a 256-bit key. If you need a refresher on AES please review chapter 4.

Listing 5.9 example payload encrypted as a JWE

```
eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..bt9LbkMg4oPbsm-l.CrF8Dq9pvvzq2grnkI99RtwUpb5geJQ-
       GdGXzWO_rVunsZr1FDmdtvzYtnV_fcf7RYdf48DrYSa5rA-
       80b_ujcv8BA9OVJ8WFWFmvyPykfAGPxiDTAuK1lR3p9CZ8Myot8DYj59UbF30-
       uLtmlUmCq6S.5vd0ehw4cKTq7iyxXqEvtQ
```

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 objected is encoded as a base64 URL string separated by a period. The JWE object shown in listing 5.9 has the initialization vector and authenticated tag bolded so you can easily spot the various components that make up a JWE.

The example JWE in listing 5.9 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 `eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0` it decodes to the JSON object

```
{
  "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 dot. The initialization vector is `bt9LbkMg4oPbsm-l` which corresponds to the random sequence bytes given the hex string `"6edf4b6e4320e283dbb26fa5"`. The ciphertext is given by the base64 URL encoded string

```
CrF8Dq9pvvzq2grnkI99RtwUpb5geJQ-GdGXzWO_rVunsZr1FDmdtvzYtnV_fcf7RYdf48DrYSa5rA-
       80b_ujcv8BA9OVJ8WFWFmvyPykfAGPxiDTAuK1lR3p9CZ8Myot8DYj59UbF30-uLtmlUmCq6S
```

The plaintext version of the JWE payload ciphertext above is the refunds.json content we have been using since chapter 2 it is shown below.

```
[ {
  "orderId" : "5555555555554444",
  "amount" : 500
}, {
  "orderId" : "4012888888881881",
  "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 url 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.

5.5.2 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. Nimbus is used by Spring Security to implement support of OpenID Connect and OAuth2 login. The JWS object explained in the previous section was created the code snippet below.

> **CODE SAMPLE** The crypto-aes-jwe sample application located at https://github.com/securing-cloud-applications/crypto-hmac-jws contains the sample code for creating and verifying JWS objects in Java. The code in the linked git repo provides detailed instructions for how to run the sample applications. I highly recommend run the sample application to deepen your understanding of the topic.

Listing 5.10 Create a JWE object encrypted with AES-GCM

```
public class CryptoUtils {

 public static String encryptAsJwe(String content, byte[] key) {
    try {
      var header = new JWEHeader(JWEAlgorithm.DIR, EncryptionMethod.A256GCM); #A
      var payload = new Payload(content); #A
      var jweObject = new JWEObject(header, payload); #A
      var aesKey = new SecretKeySpec(key, "AES"); #B
      jweObject.encrypt(new DirectEncrypter(aesKey)); #B
      return jweObject.serialize(); #C
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A create and configure the object
#B encrypt the JWE
#C serialize the JWE to a URL safe string

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 5.10 is much simpler than using the `java.crypto` library directly as we did chapter 3. 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 5.11 Decrypt a JWE object encrypted with AES-GCM

```
import com.nimbusds.jose.EncryptionMethod;
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEAlgorithm;
import com.nimbusds.jose.JWEHeader;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.DirectDecrypter;
import com.nimbusds.jose.crypto.DirectEncrypter;
import java.text.ParseException;
import javax.crypto.spec.SecretKeySpec;

public class CryptoUtils {

  public static String decryptJwe(String jwe, byte[] key) {
    try {
      JWEObject jweObject = JWEObject.parse(jwe); #A

      SecretKeySpec aesKey = new SecretKeySpec(key, "AES"); #B
      jweObject.decrypt(new DirectDecrypter(aesKey)); #B

      Payload payload = jweObject.getPayload(); #C
      return payload.toString(); #C
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A parse the JWE string into Java object
#B decrypt the JWE object
#C extract the clear text payload

The `JWEObject.parse()` method takes standard JWE base64 URL encoded string and turns it into Java object. The `DirectDecrypter` class takes care of decrypting the payload ciphertext.

Listing 5.12 Decrypt a JWE object encrypted with AES-GCM

```
import com.nimbusds.jose.EncryptionMethod;
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEAlgorithm;
import com.nimbusds.jose.JWEHeader;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.DirectDecrypter;
import com.nimbusds.jose.crypto.DirectEncrypter;
import java.text.ParseException;
import javax.crypto.spec.SecretKeySpec;

public class CryptoUtils {

  public static String decryptJwe(String jwe, byte[] key) {
    try {
      JWEObject jweObject = JWEObject.parse(jwe); #A

      SecretKeySpec aesKey = new SecretKeySpec(key, "AES");
      jweObject.decrypt(new DirectDecrypter(aesKey)); #B

      Payload payload = jweObject.getPayload(); #C
      return payload.toString();
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A parse the JWE object, the payload is still encrypted
#B decrypt the payload
#C extract the payment cleartext of the payload

5.5.3 Implementing credit cards refunds scenario using JWE

To deliver all the requirements of the credit card refunds processing scenario defined in section 3.1 we can use JWE to ensure integrity, authenticity, confidentiality using an industry standard format that is easy to implement in Java.

Listing 5.13 Generate the refunds.jwe file

```
public class RefundGenerationService {

  public void generateReport(Path refundsFile, List<Refund> refunds, byte[] key) {
    try {
      String refundsJson = JsonUtils.toJson(refunds);
      String jwe = CryptoUtils.encryptAsJwe(refundsJson, key);
      Files.writeString(refundsFile, jwe);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

The warehouse application code above is simple since we are relying on JWE the data format to make the list of refunds available to the payment service. The code listing below

shows how the payment service reads the provided refunds.jwe, checks that it was not tampered, decrypts it and processes it.

Listing 5.14 Process the refunds.jwe file

```
public class PaymentService {

  public void processRefunds(Path refundsFile, byte[] key) {
    try {
      String jwe = Files.readString(refundsFile);
      String refundsJson = CryptoUtils.decryptJwe(jwe, key);
      System.out.println("Issuing Refund to");
      System.out.println(refundsJson);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Judged against the four goals of cryptography JWE can provide integrity, authentication, and confidentiality when used in the direct encryption mode we demonstrated in this chapter. In chapter 5 we, will cover public key cryptography and see how JWS and JWE can be used with public key cryptography algorithms. In chapter 6 on digital certificates, we will learn how JWE can JWS can provide non-repudiation using public key cryptography and digital certificates.

Table 5.2 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

5.6 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. A JWT can 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. We will cover JWT in depth when cover how to implement single sign on with the OpenID connect protocol.

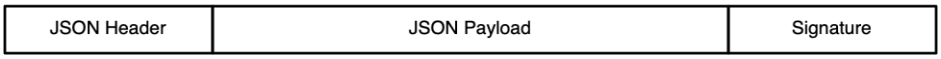


Figure 5.7 A JSON web token (JWT) can be a JWS object with the added restriction that the payload must be JSON object.

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 by the diagram below.

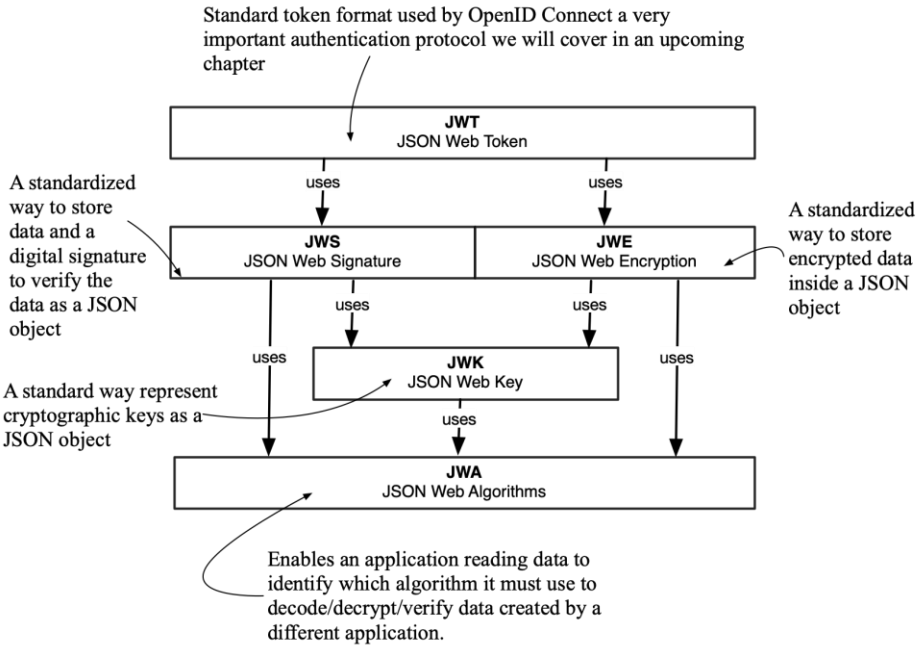


Figure 5.8 JSON Web Token (JWT) builds on top of the JWE and JWS standards. JWT is the token format for the OpenID Connect standard. The collection of standards

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 secrutiy.

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[35] 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.

5.7 Exercises

Computer security has a lot of jargon and acronyms that get used as words. It can be quite confusing to read a sentence that has 4 security acronyms in it and to figure out what it all means. A good way to get familiar with acronyms is to use them in a sentence so take a few minutes and write down an answer to following questions to get the acronyms and concepts in this chapter firmly into your mind.

1. What do JWS, JWE, JWT, JWA, and JOSE stand for?
2. Under what conditions can a JWS be considered a JWT?
3. Under what conditions can a JWE be considered a JWT?
4. What is the alg:none vulnerability in JWS and JWT how do you defend against it?
5. Is it ok to copy-and-paste the examples code in the book into production code?

5.8 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

[35] https://www.zofrex.com/blog/2020/10/20/alg-none-jwt-nhs-contact-tracing-app/

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 emphasis the concepts. Don't copy and paste the sample code blindly, you must make it production ready before you use it.

6

Public key encryption and digital signatures

This chapter covers

- Using RSA encryption with JSON Web Encryption
- Using RSA signatures with JSON Web Signature
- Using elliptic curve encryption with JSON Web Encryption
- Using elliptic curve digital signature with JSON Web Signature
- How to select a public key cryptosystem: RSA vs. elliptic curve

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

Secure communication on the internet depends on the *Transport Layer Security* (TLS) protocol. TLS is built using public key cryptography algorithms. Mastering TLS is an essential skill for a developer. If you understand public cryptography it will be easy to configure and debug TLS connections, if you don't you will be stuck copying and pasting commands from blogs hoping that whatever configuration change you made makes your application work. Public key cryptography is an indispensable tool to have in your toolbox as a developer.

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), authenticated encryption using Galois Counter Mode (GCM), JSON Web Encryption (JWE), and JSON Web Signature (JWS). 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 with an exploration of 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.

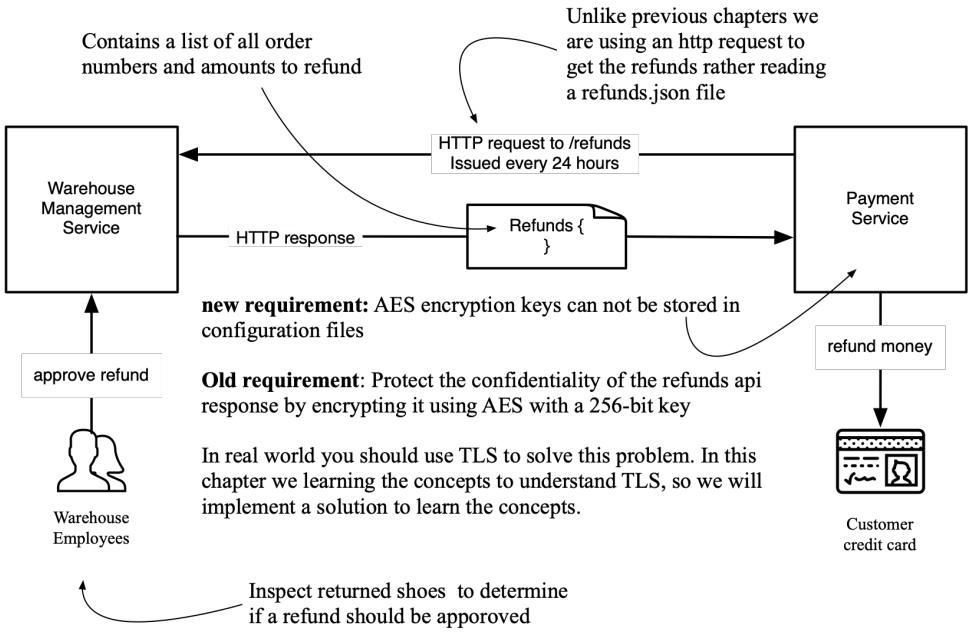


Figure 6.1 ACME Inc. staff inspect returned merchandise and approve refunds using the warehouse management service. Once per day the payment service makes an HTTP request to the warehouse management service to get a JSON array containing the order numbers and amount to refunds. The payment service issues refund to customer credit card accounts.

The warehouse management service offers a REST API endpoint that returns a list of all newly approved refunds. Once per day the payments service makes an HTTP GET request to the warehouse /refunds API end point to get a JSON object containing refunds using the response to issue refunds to customer credit cards.

Listing 6.1 Example response from the /refunds API endpoint

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

Since the `/refunds` endpoint is running over HTTP, the response must be encrypted using AES-GCM with a 256-bit key to comply with corporate security standards. Encrypting with AES requires a strategy for how to manage the secret key. In previous chapters we stored the encryption key in the Spring Boot `application.yml` configuration files of the warehouse and payment services.

Listing 6.2 application.yml file used to store AES key in pervious chapters

```
refunds:
   key: "2b6aa7fc27fdcdbb8e9c32f439efa35a"
```

Since the AES secret key is a configuration file an administrator must set the same key value in both the payments service `application.yml` and warehouse service `application.yml` file, this is not a hard thing to do if we only have two applications. However, multiple applications will need access to the `/refunds` endpoint. For example, the accounting application needs the data from `/refunds` 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. Administrators will need to set the secret key in all the applications that need to interact with the warehouse service. In an enterprise with 100s 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.yml` 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 are using in all the configuration files, across all the systems that we are running, without disrupting business operations, this is a huge effort.

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 possess very high management costs and security risks. We must find a way of sharing encryption keys without the use of configuration files. Public key cryptography provides the tools to solve the key distribution problem.

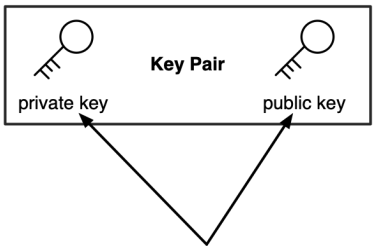
6.2 Public key cryptosystems

Thousands of years ago governments and militaries started using encryption so were faced with the key distribution problem. Over those thousands of years key distribution was done

via face-to-face meetings where keys were agreed on then memorized or couriers transported secret keys written on pieces of paper. In 1976 Whitfield Diffie and Martin Helman published a seminal paper titled "New Directions in Cryptography" introducing the world to the concept of public key cryptography and kicking off a revolution in computer security.

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. The diagram below illustrates the concept of public, private key pairs.



Private and public key are generated at the same time, are
mathematically related to each other, and only work with each other

Figure 6.2 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.

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

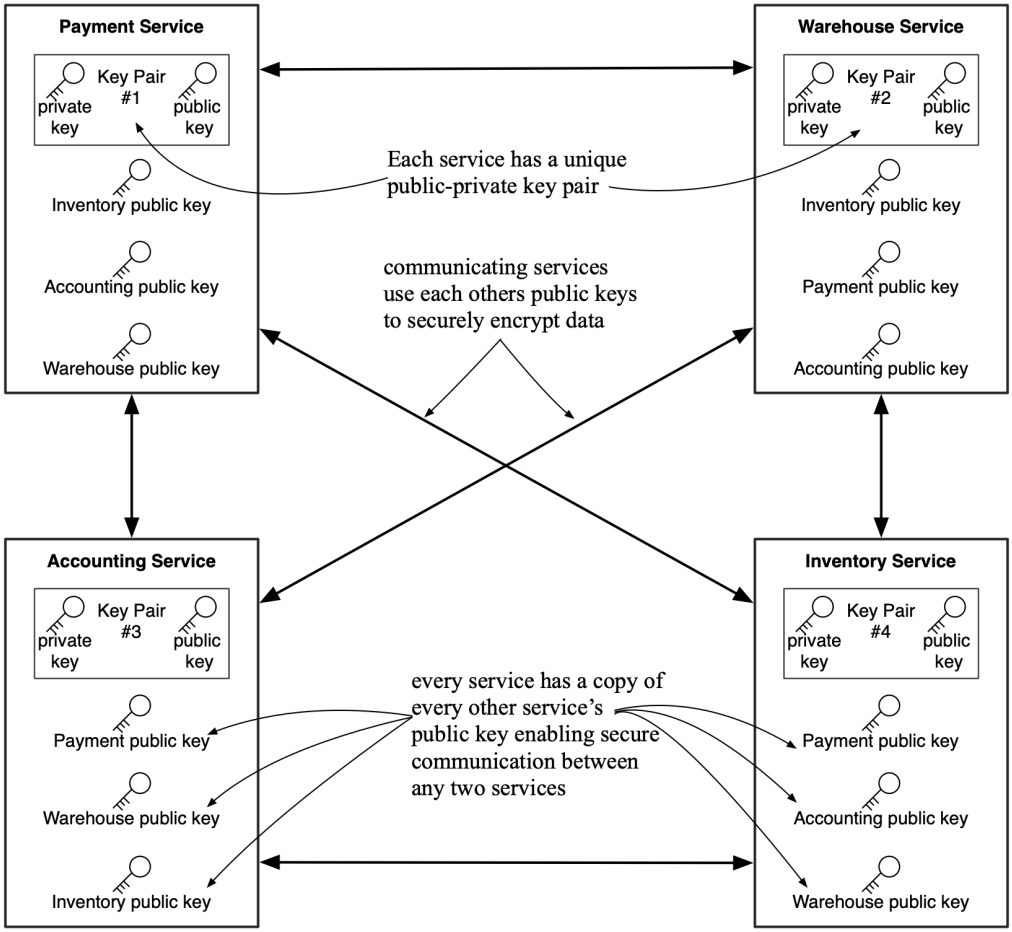


Figure 6.3 each service has a unique public-private key pair, it freely shares its public key over the network to any other service that might ask for it. When two services want to communicate, they exchange public keys and use those keys to secure the communication link. If a hacker steals a private 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. To protect against modification of a public key during transit, we encode the public key inside a digital certificate. Digital certificates are covered in the next chapter, 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 need to understand and get comfortable with digital certificates before using public key cryptography in a production application. To make the content in this chapter easier to understand we don't use digital certificates in the sample applications. Be sure to 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 and elliptic curves. The rest of this chapter provides sample applications using RSA and elliptic curves to encrypt and sign content using JSON Web Encryption (JWE) and JSON Web Signature (JWS). The samples focus on exposing important usage patterns of public key cryptography.

> **TIP** the Java cryptography libraries are low level and require a lot of attention to detail to use securely. The sample applications are built using the Nimbus Java library which provides an easy-to-use API for working with JSON Web Encryption (JWE) and JSON Web Signing (JWS) on top of the java cryptography libraries. Chapter 7 provides an introduction to Google Tink, a developer friendly crypto library for Java, if you want to avoid using JWS and JWE.

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 (https://en.wikipedia.org/wiki/RSA_numbers#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

There are two settings that must be picked when using RSA: key size and padding scheme. The larger the key size the more security you get. At the time of writing 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 minimum key length for RSA. There are two RSA based encryption schemes.

- RSAES-PKCS1-v1_5 this is an RSA encryption scheme from 1993. It is described in Public Key Cryptography Standards (PKCS) version 1.5 and RFC 2313. This mode was found to be insecure. You should never use it. Unfortunately, many products and standards concerned with backward compatibility still support RSA PCKSv1.5 exposing users to downgrade attacks. A downgrade attack is one where the attacker tricks a system into using a less secure configuration of protocol. Configure your applications to reject PCKS v1.5 so that you are protected against downgrade attacks.
- Optimal Asymmetric Encryption Padding (OAEP) introduced in 1994 resolves the security issues discovered in PCKSv1.5. You should use RSA-OAEP for applications you are working on.

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 the rest of this chapter on ECC.

6.3.2 Hybrid Encryption

The RSA encryption algorithm implementations are significantly slower than AES. Especially because processors provide special instructions for hardware accelerated AES and the mathematics for RSA encryption is CPU intensive. Therefore, it is common to combine AES with RSA into a hybrid symmetric and asymmetric scheme. The key idea is to use AES for content encryption then use 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, decryption consists of the steps below.

- 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

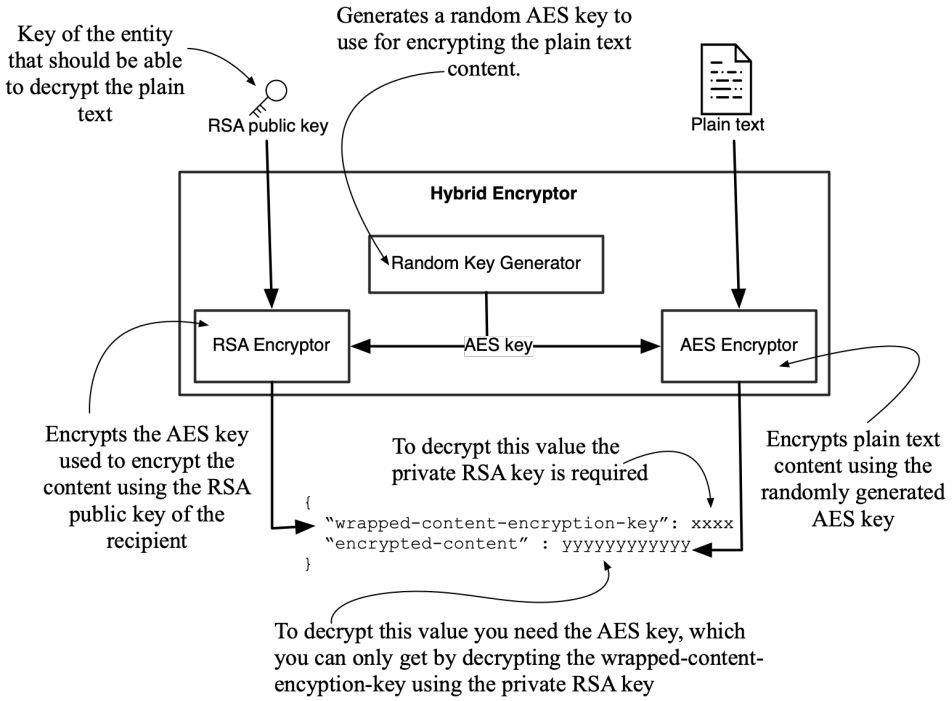


Figure 6.4 hybrid encryption uses a high-performance symmetric algorithm like AES to encrypt content. The AES Content Encryption Key (CEK) is then encrypted with the recipient's RSA public key making it possible for the recipient to use their private key to decrypt the CEK then use it to decrypt the encrypted content.

Widely deployed security protocols such as TLS, IPsec, JWE, SSH use hybrid encryption. Chapter 8,9 covers TLS and the next section 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 3 on AES for best practices on 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 Using RSA with JSON Web Encryption (JWE)

Consider a scenario where a client wants to encrypt a request so that only the server can decrypt it. The client can encrypt the request using the RSA public key of the server any hacker intercepting the request will not be able to decrypt since they don't have the server's private key. The code below shows how use JSON Web Encryption (JWE) to encrypt a payload using the hybrid encryption scheme explained in the previous section.

CODE SAMPLE the code samples in the section are part of the crypto-rsa-jwe sample application located at https://github.com/cloud-native-application-security/crypto-rsa-jwe. The code in the linked repo provides detailed instructions for how to run the sample application. I highly recommend you run the sample application to deepen your understanding of the topic.

Listing 6.3 Encrypting content using an RSA public key and JSON Web Encryption

```
import com.nimbusds.jose.EncryptionMethod;
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEAlgorithm;
import com.nimbusds.jose.JWEHeader;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.RSAEncrypter;
import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.RSAKey;
import java.text.ParseException;

public class RsaEncrypter {

  private final RSAKey key; #A

  public RsaEncryper(String publicKeyJwk) { #B
    try {
      this.key = JWK.parse(publicKeyJwk).toRSAKey();
    } catch (ParseException e) {
      throw new RuntimeException(e);
    }
  }

  public String encrypt(String data) {
    try {
      JWEHeader header = new JWEHeader(
JWEAlgorithm.RSA_OAEP_256, #C
EncryptionMethod.A256GCM); #D
      Payload payload = new Payload(data);
      JWEObject jweObject = new JWEObject(header, payload);
      jweObject.encrypt(new RSAEncrypter(key.toRSAPublicKey())); #E
      return jweObject.serialize(); #F
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A contains the RSA public key of the recipient
#B parse RSA public key encoded as a JSON Web Key (JWK) string
#C the RSA configuration used for encrypting the Content Encryption Key (CEK)
#D the AES configuration used to encrypt the content AES in GCM mode with 256-bit key
#E Nimbus RSAEncrypter requires the public key of the recipient
#F produce a JSON Web Encryption (JWE) object

The `RSAEncrpter` is a utility class is used to store an RSA public key used to encrypt content. The constructor takes the public key as a *JSON Web Key* (JWK) string and parses into an `RSAKey` object from the Nimbus Library. The encryption method uses *JSON Web Encryption*

(JWE) configured to encrypt content using a randomly generated 256-bit AES content encryption key. AES is executed in *Galois Counter Mode* (GCM) and the content encryption key is encrypted using the RSA public key using the RSA *Optimal Asymmetric Encryption Padding* (OAEP) mode. Calling the encrypt method on the input json

Listing 6.4 JSON input to encrypt into a JWE

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

produces the output JSON Web Encryption (JWE) object

Listing 6.5 JWE created from JSON in listing 6.4

```
eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiUlNBLU9BRVAtMjU2In0.DMkyTtdo-
       SZi7Wq9d9NxZU8bc0Po97NbgOGEkZk1skteStWc6Y2_KSF6uUdWDtDo2tX7Q_NN87FA81ccpGJOqZVU1jzyM
       ftlhXr0vT9TwLMJ6PChvzzh9F7lJ6iJQK4SYWVxftB_lLjhAu9SpHJRseVe1imWnsE9tdBgCCSectX0IhJ0C
       kHlLRHNBvATsvJ5NOa5V31PM-conZtLxlAKj_XwywV3y6-
       UNFsEDAG1YQHSkwpZIC8vN3qOBTRs0cGfHRqDpFwHmnN3-3pvphDZd_IGZEsSZk-
       woYS3WUOYTlT_g7EGiUHjeG6m_TJJfhw_yv_vpJbzL66tnDzWJHrR7YzPOKxkwJfyXKw76EiDEcBKb3NNLif
       wcDItQwr6mHVdqIdvod3q4DZkrxzDqUs4axlSfVDMWldE1u5LF92Rtexs0a33XaFXJNt62562Mf6B-
       LHz3I3hUkqK2QInl13vnHsjRRHGSBVjTUlLZ3TItgRXOk-
       7wZKQvDLeQ4b_JSL8w4aL3sN6b66PYRPNOp49igrUoUeGjzZik0eIx3We6L1kRzZ4JLmxjoXcRCtxx7wNGW4
       WdLUnYs5pkqqL-ob-
       hcuBw7euFF3QOsrLNTZrMkh1pxBFzMxlfO3XUETz7DPYMESVIPNmZs5GpM1LP37Y0auN4j95ZS3dyfKPYDVl
       K-I.3Jr4MRFAHwbU-
       d2f.Ec3Fwb9evT9MaCfavw1ZyDQo9IcQMYgbWI_IcA4Y217BQ8F0ww31l3u7KKuGcyLgRdYgWeq1T8RcCWm8
       1MNmp05_RbtIhciLcmgjZVVAhiQS4ttEfdGR7JIW9cpTqYgU_In8NjIfFfe8MhZQMB8MgDR1NwHESgpQ.mxS
       5iGb_b8VMb0QDWy2tVA
```

The JWE object consists of five base64url encoded parts separated by period: header, warped content encryption key, initialization vector, encrypted payload, authentication tag. The bolded content in the JWE above makes it easy to spot each of the logical parts. Review chapter if you need a refresher on JWE. To decrypt the JWE object we can use Nimbus library as show in in the listing 6.4.

List 6.5 Decrypting a JSON Web Encryption object using an RSA private key

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.crypto.RSADecrypter;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.gen.RSAKeyGenerator;
import java.text.ParseException;

public class RsaDecrypter {

  private final RSAKey key; #A
```

```
  public RsaDecrypter() { #B
    try {
      this.key = new RSAKeyGenerator(4096).generate();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }

  public String getPublicKey() { #C
    return key.toPublicJWK().toJSONString();
  }

  public String decrypt(String jwe) { #D
    try {
      JWEObject jweObject = JWEObject.parse(jwe);
      jweObject.decrypt(new RSADecrypter(key));
      return jweObject.getPayload().toString();
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A contains a private-public RSA key pair
#B generate a 4096-bit RSA key pair
#B return the public key formatted as a JSON Web Key (JWK)
#C parse the JWE and decrypt it is using the private key

The `RSADecrypter` is a utility class used to store an RSA public private key pair. The class constructor generates a 4096-bit RSA key pair which can take several seconds to compute. Since we want to be able to share the public key with others the class provides a utility method to return the public portion of the RSA key as a *JSON Web Key* (JWK). The decrypt method simply needs the string containing the JWE object and the RSA private key stored in the member variable.

In the order refunds scenario explained in section 6.1 a payment service makes an HTTP POST request to the warehouse `/refunds` service to retrieve the list of credit cards to refunds. The body of the POST requests contains the RSA public key of the payment service as shown in listing 6.5.

Listing 6.6 RSA public key represented as a JSON Web Key (JWK)

```
{
  "kty": "RSA",
  "e": "AQAB",
  "n": "oHtYPZGZwETVxhhhnVRuOUYpcjU--
       K5M32ufpdx5iiiyOjpSGoZl2k4Ou5mG4qQUMtGDZ3aJllYj3gytewM-hBXN-L54Q-
       ysOnOQQkBGly83LA1xkTa2rXOx2OcwFjyeyEM1rIpvipAAvI82GvLKNnoFstjFogTQ1-
       Y0HQYbDeCBrKTiY4CH5c_g0MypTw0_YhgfNaFKW2RzGtaEInqOpyPvkWWqFUOhAac-
       qAYz3W17W6E16A6SvBt6iCfOuSCoKfIr6zMXFxFGTG6B68nTn-Gejvo5RLW_Xvh9XUewMN4F0AZLw1FefBH-
       W_rG7ISqhIdnfuVs2mR1rxcIiq0mbs7H-aKKpE3WgLIsfxCsEbRHrucxZWAcd3Gv8wL4M4I91wuA0dG1x-
       bVB2apykTNFGdxAfsGf6IC5XPJQngM4EtotCkeAg4DrWbKhKqqGZCRLX-
       2kg1CehqUSAmVp6oGqUVNIlESuc4GzneJ5sWLxJw8_yuidbgF7L5uAV5JBKAWrEICAcX4JkrDdJqCyjeet3n
       37BNmZx2G--bFCgl4QV1rwzjtimdxk5OVMRhxYjhMrU-
       KLH2hcwpqQ92f_TIrOo7B8XQRmZMsCBgUkEl9vqSV4BYyJtJAJ-VZxWpRrsyymsasfqSs7j4UPCo_sfJ-
       XuOeMtmpaVPZqR3LZxDBTvE"
```

```
}
```

6.3.4 Man in the middle attack

The warehouse application uses the payment service's public key to encrypt the response as *a JSON Web Encryption Object* (JWE). An attacker who can capture and modify the HTTP request can perform a *Man in The Middle Attack* (MITM).

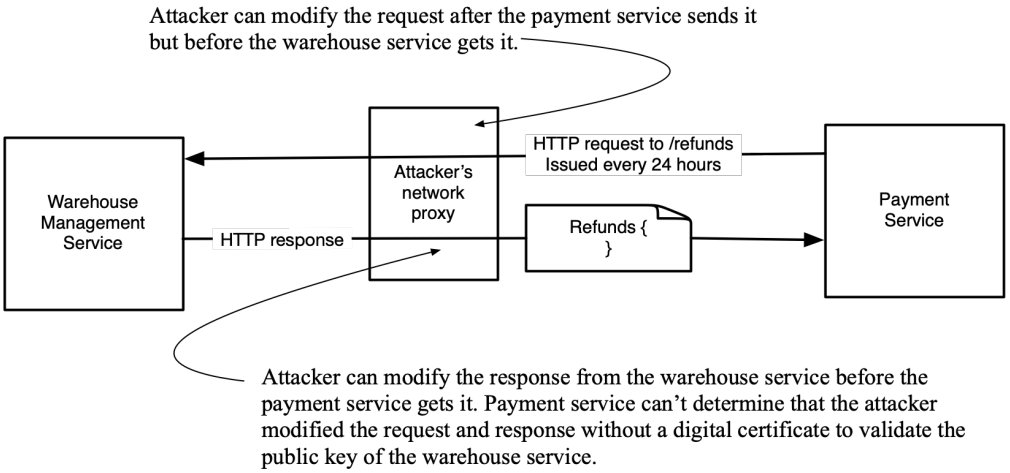


Figure 6.5 an attacker performing a man-in-the-middle (MITM) attack modifies requests and response between two communicating parties without them being able to notice the presence of the attacker. Digital certificates can be used to defeat the man in the middle attack we will cover these in the next chapter.

The steps below explain how the man in the middle attack against the warehouse and payment services works:

1. Payment Service sends a request
    a) Generate a public-private key pair
    b) Create a request body containing the public key from step a
    c) Send a request to the `/refunds` endpoint
    d) Wait for a response
2. Attacker intercepts and modifies the request
    a) Intercepts the HTTP request from the payment service
    b) Replaces the request body with the attacker's public key
    c) Forward the modified request to the warehouse service
    d) Waits for the response form warehouse service
3. Warehouse service processes the hacked request

a) Receive a request on the `/refunds` endpoint
b) Compute the response
c) Encrypt the response using the public key (attacker's key) in the request body
d) Return the encrypted response

4. Attacker intercepts modifies the response

a) Gets an encrypted response from the warehouse service
b) Decrypt the response using the attacker's private key since the warehouse service was tricked into using the attacker's public key to encrypt the response
c) Modify the response body
d) Encrypt the response body using the payment service's public key
e) Return a response to the payment service

5. Payment Service processes the modified response

a) Receive the encrypted response
b) Decrypt the encrypted response using the payment service private key everything looks legitimate, but it is not since the response was modified by the attacker.
c) Issue refunds based on modified response

To protect against the man-in-the-middle attack the warehouse service must verify that the public key it received belongs to the payment service. Validating that the public key has not been modified in transit is a hard problem that will take us a few more chapters to solve. For the rest of this chapter, focus on understanding the concepts of public key encryption with RSA so that you have a solid foundation for the upcoming discussions on how to validate the identity associated with a public key.

6.3.5 Using RSA with JSON Web Signing (JWS)

Object signing allows the receiver of a message to know that the message was not tampered with and who created the message. Anyone can read the contents of a signed message, but no one is supposed to be able to modify the message without being detected. In chapter 4 we learned how to sign messages using Hashed Message Authentication Code (HMAC). An HMAC requires a secret key to be shared between the sender the receiver. So, HMAC suffers from the same key distribution challenges that AES suffers from which we explored in section 6.1.

An RSA private key can be used to sign a message enabling the message receiver to validate the signature using the RSA public key of the sender. Since the private key should only be known to the sender, the receiver can deduce that the message must have been created by the sender and that it was not tampered with.

There are several RSA signing schemes. At the time of writing RSA Probabilistic Signature Scheme (RSA-PSS) is the recommended widely used RSA signature scheme. RSA-PSS uses a cryptographic function during the signing process. You configure which hash function to use with RSA. Typical choices are RSA-PSS with SHA-256 or SHA-384 or SHA-512. Consult with your information security team for recommendations on which RSA-PSS configuration to use. The code below shows how use JSON Web Signature (JWS) to sign a payload RSA-PSS with SHA-512.

CODE SAMPLE the code samples in the section are part of the crypto-rsa-jws sample application located at https://github.com/cloud-native-application-security/crypto-rsa-jws The code in the linked repo provides detailed instructions for how to run the sample application. I highly recommend you run the sample application to deepen your understanding of the topic.

List 6.7 Signing content using an RSA private key and JSON Web Signature

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.RSASSASigner;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.gen.RSAKeyGenerator;

public class RsaSigner {

  private final RSAKey key; #A

  public RsaSigner() { #B
    try {
      this.key = new RSAKeyGenerator(4096).generate();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }

  public String getPublicKey() { #C
    return key.toPublicJWK().toJSONString();
  }

  public String sign(String data) {
    try {
      JWSHeader header = new JWSHeader(JWSAlgorithm.PS512); #D
      Payload payload = new Payload(data);
      JWSObject jwsObject = new JWSObject(header, payload);
      jwsObject.sign(new RSASSASigner(key)); #E
      return jwsObject.serialize();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A RSA private-public key pair
#B generates a 4096-bit RSA key pair
#C return the public key as a JSON Web Key (JWK)
#D Use RSA-PSS with SHA-512 function to sign the JWS
#E Sign the JWS body

The `RSASigner` is a utility class is used to store an RSA public key used to sign content. The constructor generates a 4096-bit RSA key pair which can take several seconds to compute. Since we want to be able to share the public key with others the class provides a utility method to return the public portion of the RSA key as a JSON Web Key (JWK). The sign

method configures the JWS object to use RSA Probabilistic Signature Scheme (RSA-PSS) with SHA-512. Calling the sign method on the input json

Listing 6.8 JSON input into sign using JWS

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

produces the output JSON Web Signature (JWS) object shown below.

Listing 6.9 JWS produced from JSON input in listing 6.8

```
eyJhbGciOiJQUzUxMiJ9.WyB7CiAgImNyZWRpdENhcmQiIDogIjU1NTU1NTU1NTU1NTQ0NDQiLAogICJhbW91bnQiID
       ogNTAwCn0sIHsKICAiY3JlZGl0Q2FyZCIgOiAiNDAxMjg4ODg4ODg4MTg4MSIsCiAgImFtb3VudCIgOiAyNT
       AKfSBd.L3Vtwp45L5QZLq1uKnUqkrOSWISW34Tu3rwQH4eYejC7EWTp3K2UCPst6BEQ0dCxU4gWXKXtcEGXJ
       MBL1mPmldrO9pd9BlZgtjtId1LJRnz1ve9CI2BqgwFWrcyIiqNrYOBzM4DYS4h8-
       UUCQ7BZ2mQ9v75ZW82GkEYB8hWWsXX8T_Rfp1IDKG1uo1fxZaM3ekZmmHHOEHCdv5sDb9OQ9h0mxVlpddXCD
       d_8TC3aLymMqr97J_N3Qf1RjduLDk3L_ossIAeHTFjxBUyPMCANxNrLTnL152agSq2j_i3gLTiCrNeiMgvFP
       BPn168C2LdQLfYmBrHn1s9ECPgmVjBsdWKQv8yXKqjMDTBgEgPuvIrkHdcuG5DHnfekQ9f4xN-
       srIQCDtvW9JfNQtXMARgcTAMzcljLJeogbtUwokP7i8odkoNRZnsTUqLhnnwym0BlpANwFQI2nYiwu8ZwhsB
       79vtOTSfT9Q_jZRpZgayPf3Ltd0bk8MfU2SUJMu0eBQxluquMmWazUtwATyDvq1xsOblKLCF62e6KPB3J8oV
       sT0wN012Ye6SOoRIXlRzaU9_qjjdx6g8PBuHtCAoeOjTA_tFwatorYtuYS75xbbuJXj3ZivM_nRKoqyp0yRG
       96OXPKymG1hGQKysjWG65CW2NOfV1QMfRq8Q-dkIlAEQhjXk
```

The JWS above consists of three base64url encoded parts separated by period: header, payload, signature. The bolded content in the JWS above make it easy to spot each of the logical parts. To decrypt the JWS object we use the Nimbus library as shown in listing 6.9

List 6.9 Verifying signatures using an RSA public key and JSON Web Signature

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.crypto.RSASSAVerifier;
import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.RSAKey;
import java.text.ParseException;
import java.util.Optional;

public class RsaVerifier {

  private final RSAKey key;  #A

  public RsaVerifier(String jwk) { #B
    try {
      this.key = JWK.parse(jwk).toRSAKey();
    } catch (ParseException e) {
      throw new RuntimeException(e);
    }
  }

  public Optional<String> verify(String jws) {
    try {
```

```
      JWSObject jwsObject = JWSObject.parse(jws);
      if (jwsObject.verify(new RSASSAVerifier(key))) { #C
        return Optional.of(jwsObject.getPayload().toString());
      } else {
        return Optional.empty();
      }
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A the public key to use for verifying a signature
#B parse RSA public key encoded as a JSON Web Key (JWK) string
#C verify that the signature is valid

The `RsaVerifier` is a utility class used to store an RSA public key used to verify signed content. The constructor takes the public key as a *JSON Web Key* (JWK) string and parses into into an `RSAKey` object from the Nimbus Library. The sign method parses the JWS string and validates the signature using the stored public key. If the signature is valid the sign method returns the payload, if the signature is invalid it return an empty optional indicating to the caller that the signature was not valid.

In the credit card refunds scenario explained in section 6.1 a payment service makes an HTTP GET request to warehouse service to retrieve the public key of the payment service represented as a JSON Web Key. The payment service then makes a request to the `/refunds` endpoint on the warehouse service to retrieve the JWS object with the list of orders to issue refunds to.

An attacker who is able to capture modify and transmit the HTTP request can perform a man in the middle attack as is explained in section 6.3.4 and summarized below:

- Intercept the HTTP GET for public key of the warehouse service
- Return the public key of the attacker
- Intercept the HTTP GET response for the /refunds request
- Modify the response body
- Sign the response body using the attacker's public key
- Send the modified signed response to the payment service
- Payment service thinks that the response is valid because it has been fooled into using the attackers public key rather than the warehouse service's public key

To protect against the man-in-the-middle attack the payment service must have a way to verify that the public key it retrieved belongs to the warehouse service and not a hacker. Validating that the public key has not been modified in transit is a very hard problem that will take many more chapters to solve. For now, focus on understanding the concepts of public key signatures with RSA so that you have a solid foundation for the upcoming discussions.

6.4 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. As a developer it is important to know the following facts about the mathematics behind the ECC algorithms.

- ECC is based on the mathematical problem of computing the discrete logarithm of a point on an elliptic curve.
- 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, at the time of writing quantum computers are not powerful or widespread enough to be an immediate threat.

6.4.1 Configuring ECC

There is an infinite number of elliptic curves, the sender and receiver of a message must agree on which elliptic curve to use for encryption and decryption. Only some curves are safe to use for cryptography, so cryptographers select specific curves and give them identifying names so that security protocols can refers to curves by name rather than a mathematical equation. For example, Curve448-Goldilocks, Curve25519, P-256, secp256k1 are all commonly used curves. Configuring ECC boils down to choosing which curve to use. The following tradeoffs must be considered 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. At the time of writing 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.

6.4.2 Diffie-Helman key agreement

Elliptic curve encryption is a hybrid encryption scheme that uses AES for content encryption and *Elliptic Curve Diffie-Helman* (ECDH) algorithm to derive the AES encryption key. To understand encryption with elliptic curves we must first understand Diffie-Helman.

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.

Let's examine some network traffic to help you understand how a Diffie-Helman key exchange works. The network traffic is captured from two Spring Boot applications that implement the ACME Inc. scenario from section 6.1 illustrated in the diagram below.

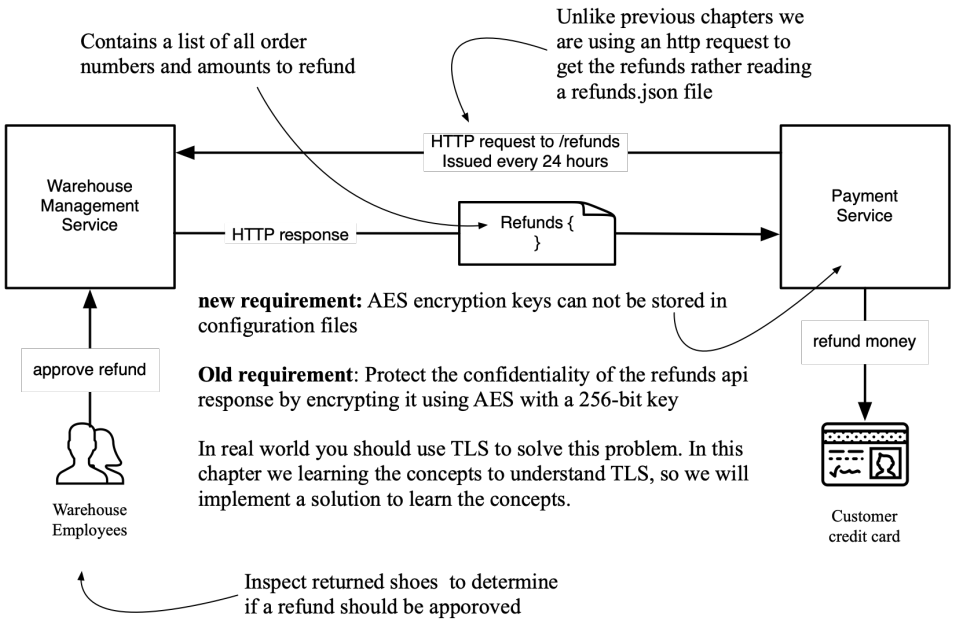


Figure 6.6 ACME Inc. staff inspect returned merchandise and approve refunds using the warehouse management service. Once a day the payment service makes an HTTP request to the warehouse management service to return a JSON array containing the credit cards and amount to refunds. The payment service issues refund to customer credit card accounts.

In the ACME Inc. scenario, the payment service makes an HTTP POST request to the warehouse service on `http://localhost:8082/refunds` which is the refunds API endpoint. The refunds API responds with AES encrypted payload. The payment service

decrypts the payload and uses it to issues refunds to customer credit cards. The AES key is computed using the Diffie-Helman key agreement algorithm. Below is step by step list of the network requests response exchanged between the payment and warehouse services.

1. Payment service generates a Diffie-Helman public-private key pair.
2. The payment service sends an HTTP POST to `http://localhost:8082/refunds` with the JSON request object containing the public key as a Base64 RL encoded string.

```
{
  "publicKey": "MCwwBwYDK2VuBQADImVC-uUECCQu4GgWvBnTjCCFBhqjwfhdKJMLByjBQ=="
}
```

3. The warehouse service receives the HTTP post request and generates its own Diffie-Helman public-private key pair.
4. The warehouse service uses the public key of the payment service and its DH key pair to compute the key `lJfPUo8haeR28OJ2NypmLYicFn2j17iMCaX3EbP0Z9w=`
5. The warehouse service generates the refunds JSON shown below.

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

6. The warehouse service encrypts the refunds JSON generated in step 5 using AES-GCM with 256-bit key `lJfPUo8haeR28OJ2NypmLYicFn2j17iMCaX3EbP0Z9w=` generated in step 4 producing the cipher text

```
eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..bNgqYsSNfGEZLIn0.O0AFHIHT5gdSYBsJuMuYkGY9pmTa_2Bwy
       NfwG14qr0HlWRi4uDmgYktSitk19lkihf9NfC-CeG9HndZ8OAskfivZgmvEnqInWknQClILqk-
       gh06XyeAextasQv68u8yq6bmdEUoEfbDSLFTre8PPodEf1lA9M7ge.wT5na8NC3axfvWAz31736Q
```

7. The warehouse returns a JSON object containing the public key of the warehouse service and the cipher text generated in step 6

```
{
  "publicKey" : "MCwwBwYDK2VuBQADIQCUwEtbvwPW8n2kpRMiJfhqWp4cQU75wWwLWvjiak-9Gg==",
  "report":
       "eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..bNgqYsSNfGEZLIn0.O0AFHIHT5gdSYBsJuMuYkGY9p
       mTa_2BwyNfwG14qr0HlWRi4uDmgYktSitk19lkihf9NfC-CeG9HndZ8OAskfivZgmvEnqInWknQClILqk-
       gh06XyeAextasQv68u8yq6bmdEUoEfbDSLFTre8PPodEf1lA9M7ge.wT5na8NC3axfvWAz31736Q"

}
```

8. The payment service receives the HTTP response containing the JSON from step 7 and reads the public key of the warehouse service
   `MCwwBwYDK2VuBQADIQCUwEtbvwPW8n2kpRMiJfhqWp4cQU75wWwLWvjiak-9Gg==`
9. The payment service uses the public key of the warehouse service and its own Diffie-Helman key pair to compute the secret key
   `lJfPUo8haeR28OJ2NypmLYicFn2j17iMCaX3EbP0Z9w=`. Notice that this is the same

value generated in step 4. Only the public keys were exchanged over the network, but both the payment service and the warehouse service were able to compute the same AES key.

10. The payment service extracts the encrypted cipher text from the response returned in step 7

```
eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..bNgqYsSNfGEZLIn0.O0AFHIHT5gdSYBs
JuMuYkGY9pmTa_2BwyNfwG14qr0HlWRi4uDmgYktSitk19lkihf9NfC-
CeG9HndZ8OAskfivZgmvEnqInWknQClILqk-
gh06XyeAextasQv68u8yq6bmdEUoEfbDSLFTre8PPodEf1lA9M7ge.wT5na8NC3axfvWAz317
36Q then decrypts it with AES-GCM using the key it computed in step 9
lJfPUo8haeR28OJ2NypmLYicFn2j17iMCaX3EbP0Z9w=
```

producing the plain text shown below

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

As you can see from the steps above, the payment service is able to make a request to the warehouse service over an insecure HTTP channel, receive an AES encrypted response, use Diffie-Helman to compute the AES encryption key then decrypt the response. The AES encryption key is never transmitted over the insecure HTTP connection, but it can be computed on both ends of the connection using Diffie-Helman key agreement algorithm. Key agreement protocols can seem like magic the diagram below provides a visualization of what a key agreement protocol.

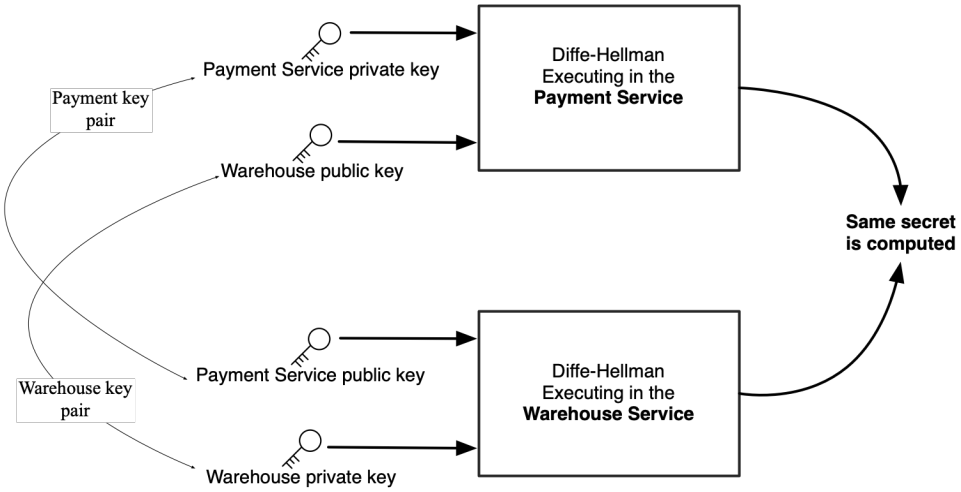


Figure 6.7 Diffie-Helman algorithm computes the same result from the (payment private key, warehouse

public key) and (warehouse private key, payment public key) key combinations.

> **TIP** you will often see code and documentation referring to Diffie-Hellman key exchange which is just another name for Diffie-Hellman 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.

> **TIP** You can find the sample application used to generate the requests in this section at https://github.com/cloud-native-application-security/crypto-key-exchange-ecdh. The sample application uses the Java 11 implementation of Elliptic Curve Diffie-Hellman using Curve25519. The AES key is derived using the HKDF function from the Google Tink library. The AES encryption is performed using the Nimbus library to produce a JWE object. Key agreement algorithms can seem like magic, I encourage you to try running the sample application and see the magic in action.

6.4.3 Using ECC with JSON Web Encryption (JWE)

The JSON Web Encryption (JWE) standard supports encrypting content using elliptic curve cryptography using a variety of curves such as Curve25519, NIST-256 and others. The example below demonstrates how to encrypt and a JWE object using Curve25519 which is considered a safe curve to use.

> **CODE SAMPLE** the code samples in the section are part of the crypto-rsa-jws sample application located at https://github.com/cloud-native-application-security/crypto-rsa-jws The code in the linked repo provides detailed instructions for how to run the sample application. I highly recommend you run the sample application to deepen your understanding of the topic.

List 6.10 Encrypting content using an ECC public key and JSON Web Encryption

```
import com.nimbusds.jose.EncryptionMethod;
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEAlgorithm;
import com.nimbusds.jose.JWEHeader;
```

```
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.X25519Encrypter;
import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.OctetKeyPair;
import java.text.ParseException;

public class EllipticCurveEncrypter {

  private final OctetKeyPair keyPair;  #A

  public EllipticCurveEncrypter(String publicKey) { #B
    try {
      this.keyPair = JWK.parse(publicKey).toOctetKeyPair();
    } catch (ParseException e) {
      throw new RuntimeException(e);
    }
  }

  public String encrypt(String data) {
    try {
      JWEHeader header = new JWEHeader(
                      JWEAlgorithm.ECDH_ES_A256KW, #C
                      EncryptionMethod.A256GCM); #D
      Payload payload = new Payload(data);
      JWEObject jweObject = new JWEObject(header, payload);
      jweObject.encrypt(new X25519Encrypter(this.keyPair)); #E
      return jweObject.serialize();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A the public key to use for encryption
#B parse ECC public key from a JSON Web Key (JWK)
#C Configure AES key wrapping
#D Configure AES content encryption
#E encrypt the content using Elliptic Curve 25519 encrypter

The `EllipticCurveEncrypter` is a utility class used to store a Curve25519 public key and use it for encryption. The class constructor takes the public key as a JSON Web Key and parses it into a Nimbus `OctetKeyPair` object. For encryption AES with a 256-bit key in GCM mode is used. The AES content encryption key is randomly generated and then encrypted using the key derived from the elliptic curve Diffie-Helman key agreement algorithm. Calling the encrypt method on the input json below

Listing 6.11 JSON input into encrypt using JWE with ECC

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
} ]
```

produces the output JSON Web Encryption (JWE) object shown below.

Listing 6.12 Encrypted JSON from input in listing 6.11

```
eyJlcGsiOnsia3R5IjoiT0tQIiwiY3J2IjoiWDI1NTE5IiwieCI6ImV1aFFGbThlNnMwS1NUNy1NcGpzZHJYbHZnLXV
       iT2tkV1ktR2dvbUlvbW8ifSwiZW5jIjoiQTI1NkdDTSIsImFsZyI6IkVDREgtRVMrQTI1NktXIn0.i-6-
       VxsKLV1wH1gLbvkzgj6dmnhNnxvXt-
       CxDIfa9WqbBnP79k0XHw.wmvUp53wHC1TF4nf.3pCpMx2ASFuuwwBh_AWOQXLnbVvAt8rOgRhs8s1_W2jHww
       Q_W7gXmmfqGpd4gcDeuJ0MC0dhB77R36tD6jxqfpv8J2Zi6LdHd4FxYZgIV-
       QJIPUIrmggWPvadBETVhw4Rd21QGHxxA9kKszt4qAbzxUJj0akjCdO.7uocXbQsc_u_MuE-WKj0yg
```

The JWE above consists of five base64url encoded parts separated by period: header, warped content encryption key, initialization vector, encrypted payload, authentication tag. The bolded content in the JWE above make it easy to spot each of the logical parts. To decrypt the JWE object we can use Nimbus library as show in in the code below.

List 6.13 Decrypting content using an ECC private key and JSON Web Encryption

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.crypto.X25519Decrypter;
import com.nimbusds.jose.jwk.Curve;
import com.nimbusds.jose.jwk.OctetKeyPair;
import com.nimbusds.jose.jwk.gen.OctetKeyPairGenerator;
import java.text.ParseException;

public class EllipticCurveDecrypter {

  private final OctetKeyPair keyPair; #A

  public EllipticCurveDecrypter() { #B
    try {
      this.keyPair = new OctetKeyPairGenerator(Curve.X25519).generate();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }

  public String getPublicKey() {  #C
    return keyPair.toPublicJWK().toJSONString();
  }

  public String decrypt(String jwe) {
    try {
      JWEObject jweObject = JWEObject.parse(jwe);
      jweObject.decrypt(new X25519Decrypter(keyPair)); #D
      return jweObject.getPayload().toString();
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A a public-private Curve25519 key pair
#B genere a new key pair
#C returns the public portion of the key pair as a JSON Web Key (JWK)
#D decrypt the JWE using the private key pair

The `EllipticCurveDecrypter` is a utility class used to store a Curve25519 key pair and can be used to decrypt content encrypted with the public key of the encrypter. The constructor generates a new X25519 key pair which is very fast compared to generating a 4096-bit RSA key pair. The decrypt method uses ecliptic curve Diffie-Helman on Curve 25519 to unwarp the content encryption key and use it to decrypt the payload.

> **TIP** elliptic curve cryptography and Diffie-Helman key exchange can be challenging to understand. Make sure to run the sample application and step through the code it will help you understand the concept.

In the credit card refunds scenario explained in section 5.1 a payment service makes an HTTP POST request to the warehouse `/refunds` endpoint to retrieve the list of credit cards to refunds. The body of the POST requests contains the Curve25119 public key of the payment service as shown below.

```
{
  "kty":"OKP",
  "crv":"X25519",
  "x":"E-dZoRCeXtsxv0oY7t3g8yUCaQUxnt5E2AxFf1f1C1U"
}
```

The warehouse application uses the payment service's public key to encrypt the resource as a JSON Web Encryption Object (JWE). An attacker who is able to capture modify and transmit the HTTP request can perform a man in the middle attack.

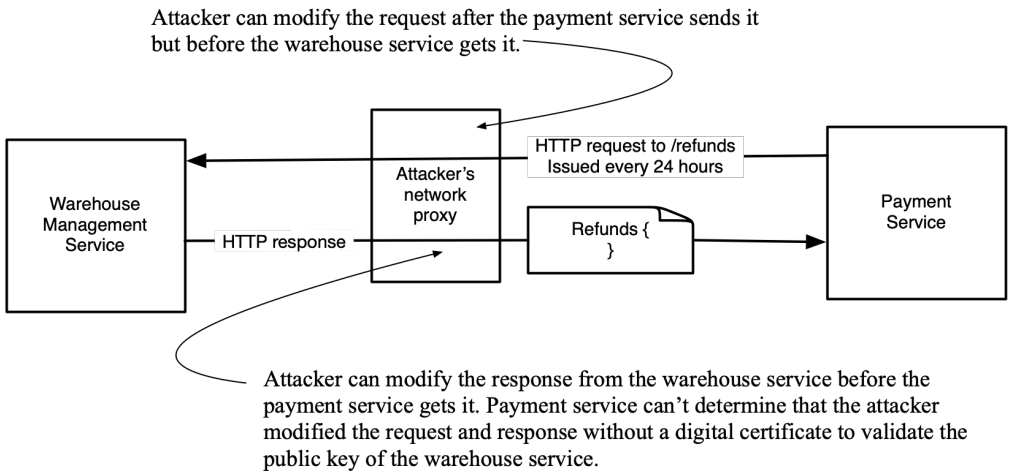


Figure 6.8 an attacker performing a man-in-the-middle (MITM) attack modifies requests and response between two communicating parties without them being able to notice the presence of the attacker. Digital certificates can be used to defeat the man in the middle attack we will cover these in the next chapter

The man in the middle attack consists of the following steps:

1. Intercept the HTTP request from the payment service
2. Replace the HTTP request body with the attacker's public key
3. Send the modified request to the warehouse service
4. Use the attacker's private key to decrypt the response from the warehouse service
5. Modify the response body
6. Encrypt the modified response body using the public key of the payment service
7. Payment service gets the encrypted response and decrypts it
8. Payment service processes the hacked refunds response without being able to tell that the request it received is not from the payment service.

To protect against the man-in-the-middle attack the warehouse service must have a way to verify that the public key it received belongs to the payment service and not to a hacker. Validating that the public key has not been modified in transit is a very hard problem that will take many more chapters to solve. For now, focus on understanding the concepts of public key encryption with RSA so that you have a solid foundation for the upcoming discussions.

6.4.4 Using ECC with JSON Web Signing (JWS)

Object signing allows the receiver of a message to know that the message was not tampered with and who created the message. Anyone can read the contents of a signed message, but no one is supposed to be able to modify the message without being detected. In chapter 4 we learned how to sign messages using Hashed Message Authentication Code (HMAC). An HMAC requires a secret key to be shared between the sender the receiver. Therefore, HMAC suffers from the same key distribution challenges that AES suffers from which we explored in section 6.1

An ECC private key can be used to sign a message enabling the message receiver to validate the signature using the ECC public key of the sender. Since the private key should only be known to the sender, the receiver can deduce that the message must have been created by the sender and that it was not tampered with.

The signing algorithm depends on the curve being used. For example, Elliptic Curve Digital Signature Algorithm (ECDSA) is used with NIST curves, while Edwards-curve Digital Signature Algorithm (EdDSA) is used with Curve 255198. As a developer you don't need to know the details of the signing algorithms, but you might need to configure your crypto library with the name of the curve to use and the associated signing algorithm. To select a safe set of ECC curves to use consult with your information security team. The code below shows how use JSON Web Signature (JWS) to sign a payload using Curve25519.

> **CODE SAMPLE** the code samples in the section are part of the crypto-ecc-jws sample application located at https://github.com/cloud-native-application-security/crypto-ecc-jws The code in the linked repo provides detailed instructions for how to run the sample application. I highly recommend you run the sample application to deepen your understanding of the topic.

List 6.14 Signing content using an Curve25519 private key and JSON Web Signature

```
import com.nimbusds.jose.JOSEException;
```

```
import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.Payload;
import com.nimbusds.jose.crypto.Ed25519Signer;
import com.nimbusds.jose.jwk.Curve;
import com.nimbusds.jose.jwk.OctetKeyPair;
import com.nimbusds.jose.jwk.gen.OctetKeyPairGenerator;

public class EllipticCurveSigner {

  private final OctetKeyPair key; #A

  public EllipticCurveSigner() { #B
    try {
      this.key = new OctetKeyPairGenerator(Curve.Ed25519).generate();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }

  public String getPublicKey() { #C
    return key.toPublicJWK().toJSONString();
  }

  public String sign(String data) {
    try {
      JWSHeader header = new JWSHeader(JWSAlgorithm.EdDSA); #D
      Payload payload = new Payload(data);
      JWSObject jwsObject = new JWSObject(header, payload);
      jwsObject.sign(new Ed25519Signer(key)); #E
      return jwsObject.serialize();
    } catch (JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A public-private ECC key pair
#B generate an ECC key pair to using Curwe 25519
#C return the public key formatted as a JSON Web Key
#D Set the JWS sigining algrothim to be Edwards-curve Digital Signature Algorithm
#E Sign the JWS using Curve 25519

The `EllipticCurverSigner` is a utility class is used to store the curve 25519 private key used to sign content. Key generation in the construction is much faster compared to generating an RSA key. Since we want to be able to share the public key with others the class provides a utility method to return the public portion of the curve 25519 key as a JSON Web Key (JWK). The sign method configures the JWS object to use Edwards-curve Digital Signature Algorithm (EdDSA). Calling the sign method on the input json below

```
[ {
  "orderId" : "12345",
  "amount" : 500
}, {
  "orderId" : "6789",
  "amount" : 250
```

```
} ]
```

produces the output JSON Web Signature (JWS) object shown below.

```
eyJhbGciOiJFZERTQSJ9.WyB7CiAgImNyZWRpdENhcmQiIDogIjU1NTU1NTU1NTU1NTQ0NDQiLAogICJhbW91bnQiID
       ogNTAwCn0sIHsKICAiY3JlZGl0Q2FyZCIgOiAiNDAxMjg4ODg4ODg4MTg4MSIsCiAgImFtb3VudCIgOiAyNT
       AKfSBd.hl6xtGsbdshMNV4c9unUP5tqsNcBTBdEGa9t6syRJ5zZYIgBhC2i0QtkpO7dLo_Ur-
       uV2elmVDuGs3vissXoDA
```

The JWS above consists of three base64url encoded parts separated by period: header, payload, signature. The bolded content in the JWS above make it easy to spot each of the logical parts. To decrypt the JWS object we can use Nimbus library as show in in the code below.

List 6.15 Verifying signatures using a curve 25519 public key and JSON Web Signature

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWSObject;
import com.nimbusds.jose.crypto.Ed25519Verifier;
import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.OctetKeyPair;
import java.text.ParseException;
import java.util.Optional;

public class EllipticCurveVerifier {

  private final OctetKeyPair key; #A

  public EllipticCurveVerifier(String publicKey) { #B
    try {
      this.key = JWK.parse(publicKey).toOctetKeyPair();
    } catch (ParseException e) {
      throw new RuntimeException(e);
    }
  }

  public Optional<String> verify(String jws) {
    try {
      JWSObject jwsObject = JWSObject.parse(jws);
      if (jwsObject.verify(new Ed25519Verifier(key))) {  #C
        return Optional.of(jwsObject.getPayload().toString());
      } else {
        return Optional.empty();
      }
    } catch (ParseException | JOSEException e) {
      throw new RuntimeException(e);
    }
  }
}
```

#A the ECC public key to use for verifying a signature
#B parse public key passed as a JSON Web Key
#C Verify the JWS object using the Edwards-curve Digital Signature Algorithm

The EllipticCurveVerifier is a utility class used to store an curve 25519 public key used to verify signed content. The constructor takes the public key as a JSON Web Key (JWK) string and parses into into an OctetPair object from the Nimbus Library. The verify method parses the JWS string and validates the signature using the stored public key. If the signature is valid the sign method returns the payload, if the signature is invalid it return an empty optional indicating to the caller that the signature was not valid.

In the credit card refunds scenario explained in section 5.1 a payment service makes an HTTP GET request to warehouse service to retrieve the public key of the payment service represented as a JSON Web Key. The payment service then makes a request to the /refunds endpoints on the warehouse service to retrieve the JWS object with the list of credit cards to issue refunds to. An attacker who is able to capture modify and transmit the HTTP request can perform the following man in the middle attack.

- Intercept the HTTP GET for public key of the warehouse service
- Return the public key of the attacker
- Intercept the HTTP GET response for the /refunds request
- Modify the response body
- Sign the response body using the attacker's public key
- Send the modified signed response to the payment service
- Payment service thinks that the response is valid because it has been fooled into using the attackers public key rather than the warehouse service's public key

To protect against the man-in-the-middle attack the payment service must have a way to verify that the public key it retrieved belongs to the warehouse service and not a hacker. Validating that the public key has not been modified in transit is a very hard problem that will take many more chapters to solve. For now, focus on understanding the concepts of public key signatures with ECC so that you have a solid foundation for the upcoming discussions.

6.5 RSA vs. ECC

At the time of writing 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 libraires. 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 recommend RSA and ECC configurations.

RSA and ECC are based on mathematical problems that are 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 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 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.

At the time of writing 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.

6.6 Summary

- 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.
- ECC has better performance than RSA and most cryptographers recommend using ECC over RSA for new applications.
- Quantum computers will break RSA and ECC because they solve the mathematical problems used by RSA and ECC 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.
- 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 encrypting using RSA or ECC.
- JSON Web Encryption (JWE) supports RSA and ECC encryption.
- JSON Web Signature (JWE) supports RSA and ECC signing.
- 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 emphasis the concepts Don't copy and paste the sample code blindly, you must make it production ready before you use it.