1 What is API security?

This chapter covers

- What is an API?
- What makes an API secure or insecure?
- Defining security in terms of goals
- Identifying threats and vulnerabilities
- Using mechanisms to achieve security goals

Application Programming Interfaces (APIs) are everywhere. Open your smartphone or tablet and look at the apps you have installed. Almost without exception, those apps are talking to one or more remote APIs to download fresh content and messages, poll for notifications, upload your new content, and perform actions on your behalf.

Load your favorite web page with the developer tools open in your browser, and you'll likely see dozens of API calls happening in the background to render a page that is heavily customized to you as an individual (whether you like it or not). On the server, those API calls may themselves be implemented by many microservices communicating with each other via internal APIs.

Increasingly, even the everyday items in your home are talking to APIs in the cloud--from smart speakers like Amazon Echo or Google Home, to refrigerators, electricity meters, and lightbulbs. The Internet of Things (IoT) is rapidly becoming a reality in both consumer and industrial settings, powered by ever-growing numbers of APIs in the cloud and on the devices themselves.

While the spread of APIs is driving ever more sophisticated applications that enhance and amplify our own abilities, they also bring increased risks. As we become more dependent on APIs for critical tasks in work and play, we become more vulnerable if they are attacked. The more APIs are used, the greater their potential to be attacked. The very property that

makes APIs attractive for developers--ease of use--also makes them an easy target for malicious actors. At the same time, new privacy and data protection legislation, such as the GDPR in the EU, place legal requirements on companies to protect users' data, with stiff penalties if data protections are found to be inadequate.

GDPR

The General Data Protection Regulation (GDPR) is a significant piece of EU law that came into force in 2018. The aim of the law is to ensure that EU citizens' personal data is not abused and is adequately protected by both technical and organizational controls. This includes security controls that will be covered in this book, as well as privacy techniques such as pseudonymization of names and other personal information (which we will not cover) and requiring explicit consent before collecting or sharing personal data. The law requires companies to report any data breaches within 72 hours and violations of the law can result in fines of up to €20 million (approximately $23.6 million) or 4% of the worldwide annual turnover of the company. Other jurisdictions are following the lead of the EU and introducing similar privacy and data protection legislation.

This book is about how to secure your APIs against these threats so that you can confidently expose them to the world.

1.1 An analogy: Taking your driving test

To illustrate some of the concepts of API security, consider an analogy from real life: taking your driving test. This may not seem at first to have much to do with either APIs or security, but as you will see, there are similarities between aspects of this story and key concepts that you will learn in this chapter.

You finish work at 5 p.m. as usual. But today is special. Rather than going home to tend to your carnivorous plant collection and then flopping down in front of the TV, you have somewhere else to be. Today you are taking your driving test.

You rush out of your office and across the park to catch a bus to the test center. As you stumble past the queue of people at the hot dog stand, you see your old friend Alice walking her pet alpaca, Horatio.

“Hi Alice!” you bellow jovially. “How’s the miniature recreation of 18th-century Paris coming along?”

“Good!” she replies. “You should come and see it soon.”

She makes the universally recognized hand-gesture for “call me” and you both hurry on your separate ways.

You arrive at the test center a little hot and bothered from the crowded bus journey. If only you could drive, you think to yourself! After a short wait, the examiner comes out and introduces himself. He asks to see your learner’s driving license and studies the old photo of you with that bad haircut you thought was pretty cool at the time. After a few seconds of quizzical stares, he eventually accepts that it is really you, and you can begin the test.

LEARN ABOUT IT Most APIs need to identify the clients that are interacting with them. As these fictional interactions illustrate, there may be different ways of identifying your API clients that are appropriate in different situations. As with Alice, sometimes there is a long-standing trust relationship based on a history of previous interactions, while in other cases a more formal proof of identity is required, like showing a driving license. The examiner trusts the license because it is issued by a trusted body, and you match the photo on the license. Your API may allow some operations to be performed with only minimal identification of the user but require a higher level of identity assurance for other operations.

You failed the test this time, so you decide to take a train home. At the station you buy a standard class ticket back to your suburban neighborhood, but feeling a little devil-may-care, you decide to sneak into the first-class carriage. Unfortunately, an attendant blocks your way and demands to see your ticket. Meekly you scurry back into standard class and slump into your seat with your headphones on.

When you arrive home, you see the light flashing on your answering machine. Huh, you’d forgotten you even had an answering machine. It’s Alice, inviting you to the hot new club that just opened in town. You could do with a night out to cheer you up, so you decide to go.

The doorwoman takes one look at you.

“Not tonight,” she says with an air of sniffy finality.

At that moment, a famous celebrity walks up and is ushered straight inside. Dejected and rejected, you head home.

What you need is a vacation. You book yourself a two-week stay in a fancy hotel. While you are away, you give your neighbor Bob the key to your tropical greenhouse so that he can feed your carnivorous plant collection. Unknown to you, Bob throws a huge party in your back garden and invites half the town. Thankfully, due to a miscalculation, they run out of drinks before any real damage is done (except to Bob’s reputation) and the party disperses. Your prized whisky selection remains safely locked away inside.

LEARN ABOUT IT Beyond just identifying your users, an API also needs to be able to decide what level of access they should have. This can be based on who they are, like the celebrity getting into the club, or based on a limited-time token like a train ticket, or a long-term key like the key to the greenhouse that you lent your neighbor. Each approach has different trade-offs. A key can be lost or stolen and then used by anybody. On the other hand, you can have different keys for different locks (or different operations) allowing only a small amount of authority to be given to somebody else. Bob could get into the greenhouse and garden but not into your house and whisky collection.

When you return from your trip, you review the footage from your comprehensive (some might say over-the-top) camera surveillance system. You cross Bob off the Christmas card list and make a mental note to ask someone else to look after the plants next time.

The next time you see Bob you confront him about the party. He tries to deny it at first, but when you point out the cameras, he admits everything. He buys you a lovely new Venus flytrap to say sorry. The video cameras show the advantage of having good audit logs so that you can find out who did what when things go wrong, and if necessary, prove who was responsible in a way they cannot easily deny.

DEFINITION An audit log records details of significant actions taken on a system, so that you can later work out who did what and when. Audit logs are crucial evidence when investigating potential security breaches.

You can hopefully now see a few of the mechanisms that are involved in securing an API, but before we dive into the details let's review what an API is and what it means for it to be secure.

1.2 What is an API?

Traditionally, an API was provided by a software library that could be linked into an application either statically or dynamically at runtime, allowing reuse of procedures and functions for specific problems, such as OpenGL for 3D graphics, or libraries for TCP/IP networking. Such APIs are still common, but a growing number of APIs are now made available over the internet as RESTful web services.

Broadly speaking, an API is a boundary between one part of a software system and another. It defines a set of operations that one component provides for other parts of the system (or other systems) to use. For example, a photography archive might provide an API to list albums of photos, to view individual photos, add comments, and so on. An online image gallery could then use that API to display interesting photos, while a word processor application could use the same API to allow embedding images into a document. As shown in figure 1.1, an API handles requests from one or more clients on behalf of users. A client may be a web or mobile application with a user interface (UI), or it may be another API with no explicit UI. The API itself may talk to other APIs to get its work done.

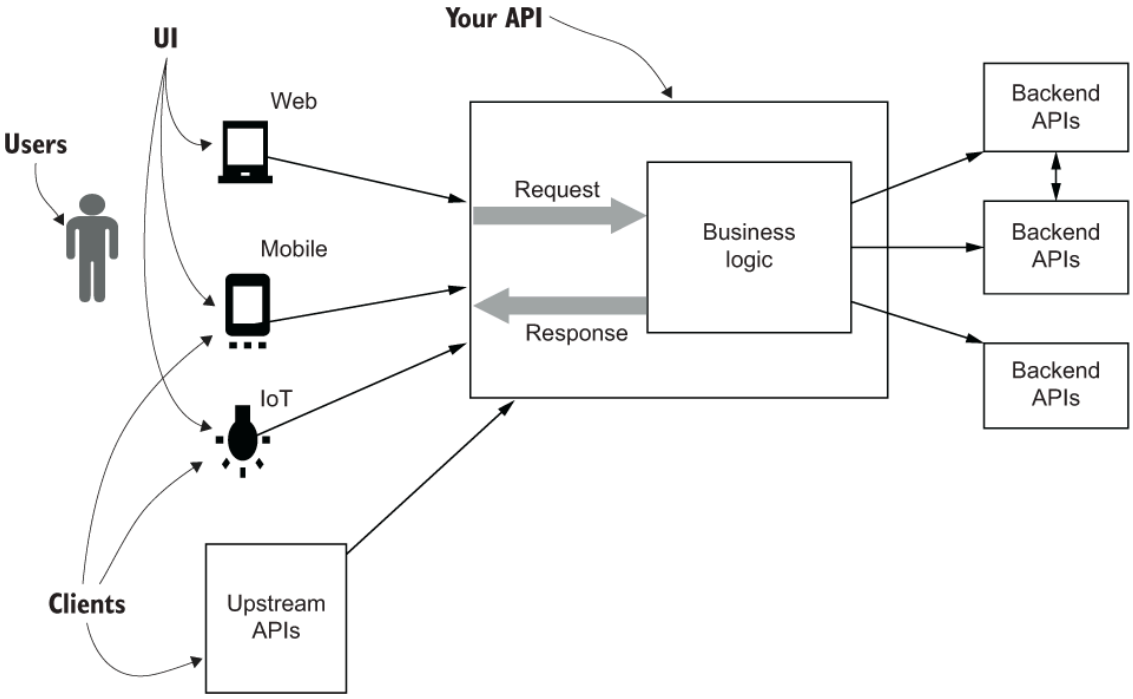

Figure 1.1 An API handles requests from clients on behalf of users. Clients may be web browsers, mobile apps, devices in the Internet of Things, or other APIs. The API services requests according to its internal logic and then at some point returns a response to the client. The implementation of the API may require talking to other "backend" APIs, provided by databases or processing systems.

A UI also provides a boundary to a software system and restricts the operations that can be performed. What distinguishes an API from a UI is that an API is explicitly designed to be easy to interact with by other software, while a UI is designed to be easy for a user to interact with directly. Although a UI might present information in a rich form to make the information pleasing to read and easy to interact with, an API typically will present instead a highly regular and stripped-back view of the raw data in a form that is easy for a program to parse and manipulate.

1.2.1 API styles

There are several popular approaches to exposing remote APIs:

- Remote Procedure Call (RPC) APIs expose a set of procedures or functions that can be called by clients over a network connection. The RPC style is designed to resemble normal procedure calls as if the API were provided locally. RPC APIs often use compact binary formats for messages and are very efficient, but usually require the client to install specific libraries (known as stubs) that work with a single API. The gRPC framework from Google (**https://grpc.io**) is an example of a modern RPC approach. The older SOAP (Simple Object Access Protocol) framework, which uses XML for messages, is still widely deployed.
- A variant of the RPC style known as Remote Method Invocation (RMI) uses object-oriented techniques to allow clients to call methods on remote objects as if they were local. RMI approaches used to be very popular, with technologies such as CORBA and Enterprise Java Beans (EJBs) often used for building large enterprise systems. The complexity of these frameworks has led to a decline in their use.
- The REST (REpresentational State Transfer) style was developed by Roy Fielding to describe the principles that led to the success of HTTP and the web and was later adapted as a set of principles for API design. In contrast to RPC, RESTful APIs emphasize standard message formats and a small number of generic operations to reduce the coupling between a client and a specific API. Use of hyperlinks to navigate the API reduce the risk of clients breaking as the API evolves over time.
- Some APIs are mostly concerned with efficient querying and filtering of large data sets, such as SQL databases or the GraphQL framework from Facebook (**https://graphql.org**). In these cases, the API often only provides a few operations and a complex query language allows the client significant control over what data is returned.

Different API styles are suitable for different environments. For example, an organization that has adopted a microservices architecture might opt for an efficient RPC framework to reduce the overhead of API calls. This is appropriate because the organization controls all of the clients and servers in this environment and can manage distributing new stub libraries when they are required. On the other hand, a widely used public API might be better suited to the REST style using a widely used format such as JSON to maximize interoperability with different types of clients.

DEFINITION In a microservices architecture, an application is deployed as a collection of loosely coupled services rather than a single large application, or monolith. Each microservice exposes an API that other services talk to. Securing microservice APIs is covered in detail in part 4 of this book.

This book will focus on APIs exposed over HTTP using a loosely RESTful approach, as this is the predominant style of API at the time of writing. That is, although the APIs that are developed in this book will try to follow REST design principles, you will sometimes deviate from those principles to demonstrate how to secure other styles of API design. Much of the advice will apply to other styles too, and the general principles will even apply when designing a library.

1.3 API security in context

API Security lies at the intersection of several security disciplines, as shown in figure 1.2. The most important of these are the following three areas:

1. Information security (InfoSec) is concerned with the protection of information over its full life cycle from creation, storage, transmission, backup, and eventual destruction.
2. Network security deals with both the protection of data flowing over a network and prevention of unauthorized access to the network itself.
3. Application security (AppSec) ensures that software systems are designed and built to withstand attacks and misuse.

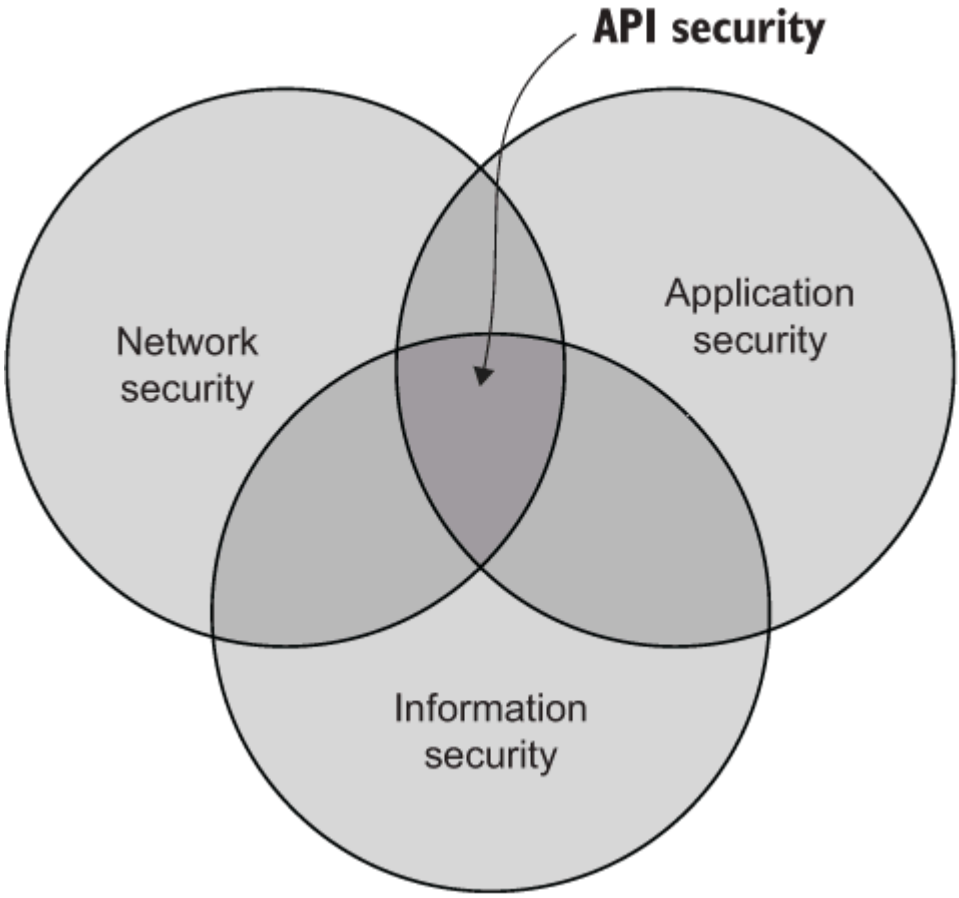

Figure 1.2 API security lies at the intersection of three security areas: information security, network security, and application security.

Each of these three topics has filled many books individually, so we will not cover each of them in full depth. As figure 1.2 illustrates, you do not need to learn every aspect of these topics to know how to build secure APIs. Instead, we will pick the most critical areas from each and blend them to give you a thorough understanding of how they apply to securing an API.

From information security you will learn how to:

- Define your security goals and identify threats
- Protect your APIs using access control techniques
- Secure information using applied cryptography

DEFINITION Cryptography is the science of protecting information so that two or more people can communicate without their messages being read or tampered with by anybody else. It can also be used to protect information written to disk.

From network security you will learn:

- The basic infrastructure used to protect an API on the internet, including firewalls, load-balancers, and reverse proxies, and roles they play in protecting your API (see the next section)
- Use of secure communication protocols such as HTTPS to protect data transmitted to or from your API

DEFINITION HTTPS is the name for HTTP running over a secure connection. While normal HTTP requests and responses are visible to anybody watching the network traffic, HTTPS messages are hidden and protected by Transport Layer Security (TLS, also known as SSL). You will learn how to enable HTTPS for an API in chapter 3.

Finally, from application security you will learn:

- Secure coding techniques
- Common software security vulnerabilities
- How to store and manage system and user credentials used to access your APIs

1.3.1 A typical API deployment

An API is implemented by application code running on a server; either an application server such as Java Enterprise Edition (Java EE), or a standalone server. It is very rare to directly expose such a server to the internet, or even to an internal intranet. Instead, requests to the API will typically pass through one or more additional network services before they reach your API servers, as shown in figure 1.3. Each request will pass through one or more firewalls, which inspect network traffic at a relatively low level and ensure that any unexpected traffic is blocked. For example, if your APIs are serving requests on port 80 (for HTTP) and 443 (for HTTPS), then the firewall would be configured to block any requests for any other ports. A load balancer will then route traffic to appropriate services and ensure that one server is not overloaded with lots of requests while others sit idle. Finally, a reverse proxy (or gateway) is typically placed in front of the application servers to perform computationally expensive operations like handling TLS encryption (known as SSL termination) and validating credentials on requests.

DEFINITION SSL termination[1] (or SSL offloading) occurs when a TLS connection from a client is handled by a load balancer or reverse proxy in

front of the destination API server. A separate connection from the proxy to the backend server is then made, which may either be unencrypted (plain HTTP) or encrypted as a separate TLS connection (known as SSL re-encryption).

Beyond these basic elements, you may encounter several more specialist services:

- An API gateway is a specialized reverse proxy that can make different APIs appear as if they are a single API. They are often used within a microservices architecture to simplify the API presented to clients. API gateways can often also take care of some of the aspects of API security discussed in this book, such as authentication or rate-limiting.
- A web application firewall (WAF) inspects traffic at a higher level than a traditional firewall and can detect and block many common attacks against HTTP web services.
- An intrusion detection system (IDS) or intrusion prevention system (IPS) monitors traffic within your internal networks. When it detects suspicious patterns of activity it can either raise an alert or actively attempt to block the suspicious traffic.

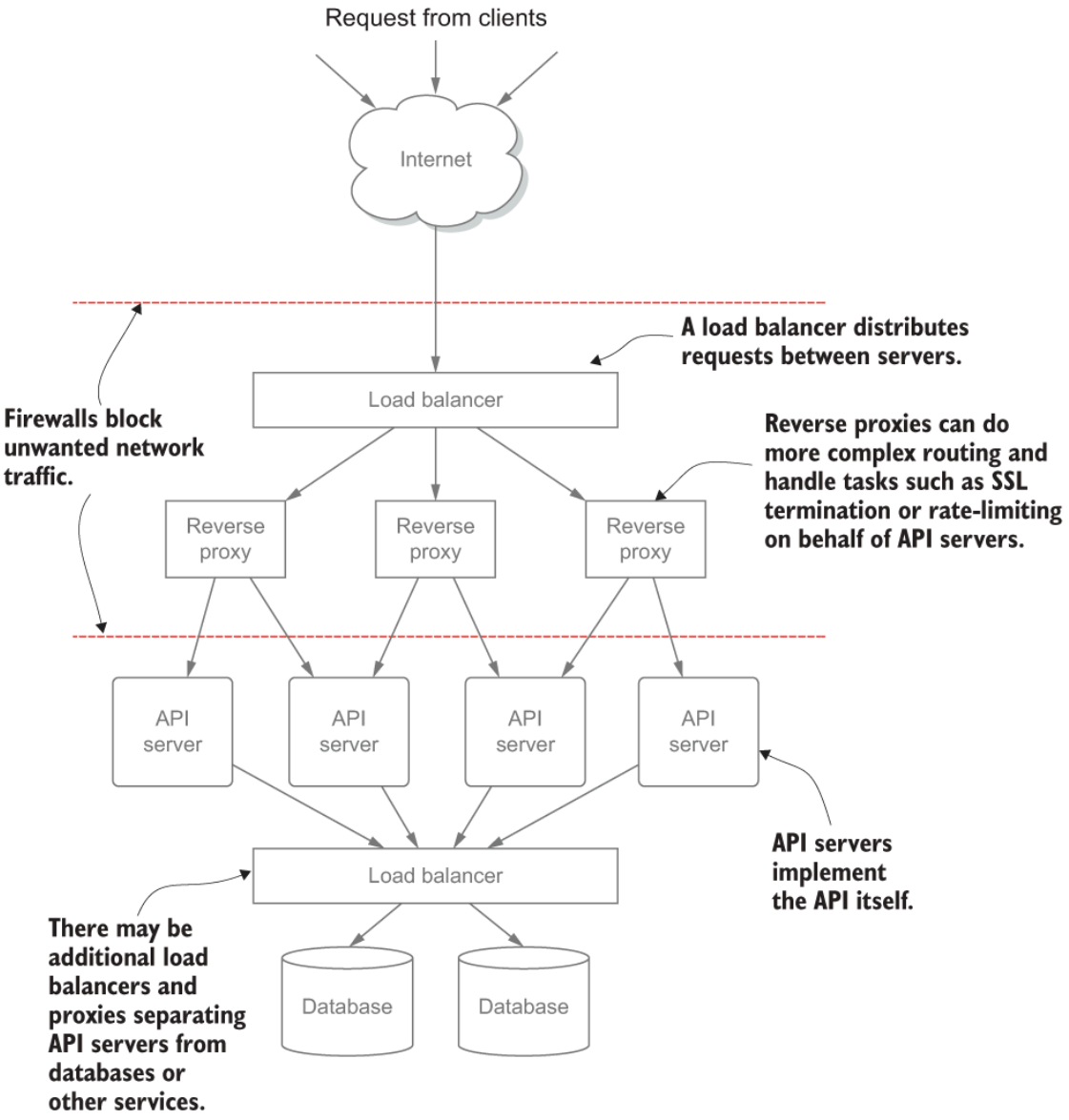

Figure 1.3 Requests to your API servers will typically pass through several other services first. A firewall works at the TCP/IP level and only allows traffic in or out of the network that matches expected flows. A load balancer routes requests to appropriate internal services based on the request and on its knowledge of how much work each server is currently doing. A reverse proxy or API gateway can take care of expensive tasks on behalf of the API server, such as terminating HTTPS connections or validating authentication credentials.

In practice, there is often some overlap between these services. For example, many load balancers are also capable of performing tasks of a reverse proxy, such as terminating TLS connections, while many reverse proxies can also function as an API gateway. Certain more specialized services can even handle many of the security mechanisms that you will learn in this book, and it is becoming common to let a gateway or reverse proxy handle at least some of these tasks. There are limits to what these components can do, and poor security practices in your APIs can under-

mine even the most sophisticated gateway. A poorly configured gateway can also introduce new risks to your network. Understanding the basic security mechanisms used by these products will help you assess whether a product is suitable for your application, and exactly what its strengths and limitations are.

Pop quiz

1. Which of the following topics are directly relevant to API security? (Select all that apply.)
    1. Job security
    2. National security
    3. Network security
    4. Financial security
    5. Application security
    6. Information security
2. An API gateway is a specialized version of which one of the following components?
    1. Client
    2. Database
    3. Load balancer
    4. Reverse proxy
    5. Application server

The answers are at the end of the chapter.

1.4 Elements of API security

An API by its very nature defines a set of operations that a caller is permitted to use. If you don't want a user to perform some operation, then simply exclude it from the API. So why do we need to care about API security at all?

- First, the same API may be accessible to users with distinct levels of authority; for example, with some operations allowed for only administrators or other users with a special role. The API may also be exposed to users (and bots) on the internet who shouldn't have any access at all. Without appropriate access controls, any user can perform any action, which is likely to be undesirable. These are factors related to the environment in which the API must operate.

- Second, while each individual operation in an API may be secure on its own, combinations of operations might not be. For example, a banking API might offer separate withdrawal and deposit operations, which individually check that limits are not exceeded. But the deposit operation has no way to know if the money being deposited has come from a real account. A better API would offer a transfer operation that moves money from one account to another in a single operation, guaranteeing that the same amount of money always exists. The security of an API needs to be considered as a whole, and not as individual operations.
- Last, there may be security vulnerabilities due to the implementation of the API. For example, failing to check the size of inputs to your API may allow an attacker to bring down your server by sending a very large input that consumes all available memory; a type of denial of service (DoS) attack.

DEFINITION A denial of service (DoS) attack occurs when an attacker can prevent legitimate users from accessing a service. This is often done by flooding a service with network traffic, preventing it from servicing legitimate requests, but can also be achieved by disconnecting network connections or exploiting bugs to crash the server.

Some API designs are more amenable to secure implementation than others, and there are tools and techniques that can help to ensure a secure implementation. It is much easier (and cheaper) to think about secure development before you begin coding rather than waiting until security defects are identified later in development or in production. Retrospectively altering a design and development life cycle to account for security is possible, but rarely easy. This book will teach you practical techniques for securing APIs, but if you want a more thorough grounding in how to design-in security from the start, then I recommend the book Secure by Design by Dan Bergh Johnsson, Daniel Deogun, and Daniel Sawano (Manning, 2019).

It is important to remember that there is no such thing as a perfectly secure system, and there is not even a single definition of “security.” For a healthcare provider, being able to discover whether your friends have accounts on a system would be considered a major security flaw and a privacy violation. However, for a social network, the same capability is an essential feature. Security therefore depends on the context. There are

many aspects that should be considered when designing a secure API, including the following:

- The assets that are to be protected, including data, resources, and physical devices
- Which security goals are important, such as confidentiality of account names
- The mechanisms that are available to achieve those goals
- The environment in which the API is to operate, and the threats that exist in that environment

1.4.1 Assets

For most APIs, the assets will consist of information, such as customer names and addresses, credit card information, and the contents of databases. If you store information about individuals, particularly if it may be sensitive such as sexual orientation or political affiliations, then this information should also be considered an asset to be protected.

There are also physical assets to consider, such as the physical servers or devices that your API is running on. For servers running in a datacenter, there are relatively few risks of an intruder stealing or damaging the hardware itself, due to physical protections (fences, walls, locks, surveillance cameras, and so on) and the vetting and monitoring of staff that work in those environments. But an attacker may be able to gain control of the resources that the hardware provides through weaknesses in the operating system or software running on it. If they can install their own software, they may be able to use your hardware to perform their own actions and stop your legitimate software from functioning correctly.

In short, anything connected with your system that has value to somebody should be considered an asset. Put another way, if anybody would suffer real or perceived harm if some part of the system were compromised, that part should be considered an asset to be protected. That harm may be direct, such as loss of money, or it may be more abstract, such as loss of reputation. For example, if you do not properly protect your users' passwords and they are stolen by an attacker, the users may suffer direct harm due to the compromise of their individual accounts, but your organization would also suffer reputational damage if it became known that you hadn't followed basic security precautions.

1.4.2 Security goals

Security goals are used to define what security actually means for the protection of your assets. There is no single definition of security, and some definitions can even be contradictory! You can break down the notion of security in terms of the goals that should be achieved or preserved by the correct operation of the system. There are several standard security goals that apply to almost all systems. The most famous of these are the so-called “CIA Triad”:

- Confidentiality --Ensuring information can only be read by its intended audience
- Integrity --Preventing unauthorized creation, modification, or destruction of information
- Availability --Ensuring that the legitimate users of an API can access it when they need to and are not prevented from doing so.

Although these three properties are almost always important, there are other security goals that may be just as important in different contexts, such as accountability (who did what) or non-repudiation (not being able to deny having performed an action). We will discuss security goals in depth as you develop aspects of a sample API.

Security goals can be viewed as non-functional requirements (NFRs) and considered alongside other NFRs such as performance or reliability goals. In common with other NFRs, it can be difficult to define exactly when a security goal has been satisfied. It is hard to prove that a security goal is never violated because this involves proving a negative, but it’s also difficult to quantify what “good enough” confidentiality is, for example.

One approach to making security goals precise is used in cryptography. Here, security goals are considered as a kind of game between an attacker and the system, with the attacker given various powers. A standard game for confidentiality is known as indistinguishability. In this game, shown in figure 1.4, the attacker gives the system two equal-length messages, A and B, of their choosing and then the system gives back the encryption of either one or the other. The attacker wins the game if they can determine which of A or B was given back to them. The system is said to be secure (for this security goal) if no realistic attacker has better than a 50:50 chance of guessing correctly.

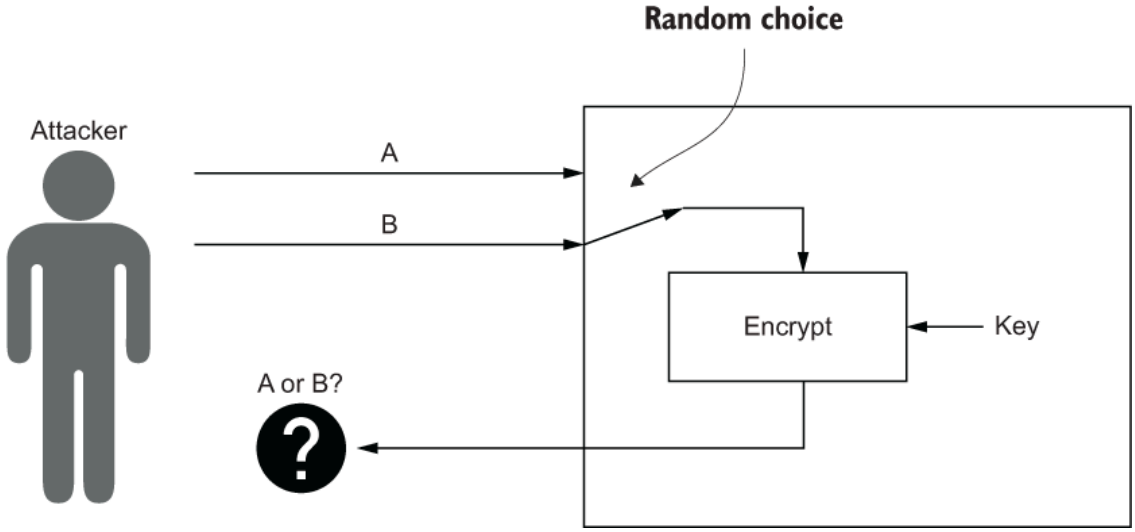

Figure 1.4 The indistinguishability game used to define confidentiality in cryptography. The attacker is allowed to submit two equal-length messages, A and B. The system then picks one at random and encrypts it using the key. The system is secure if no "efficient" challenger can do much better than guesswork to know whether they received the encryption of message A or B.

Not every scenario can be made as precise as those used in cryptography. An alternative is to refine more abstract security goals into specific requirements that are concrete enough to be testable. For example, an instant messaging API might have the functional requirement that users are able to read their messages. To preserve confidentiality, you may then add constraints that users are only able to read their own messages and that a user must be logged in before they can read their messages. In this approach, security goals become constraints on existing functional requirements. It then becomes easier to think up test cases. For example:

- Create two users and populate their accounts with dummy messages.
- Check that the first user cannot read the messages of the second user.
- Check that a user that has not logged in cannot read any messages.

There is no single correct way to break down a security goal into specific requirements, and so the process is always one of iteration and refinement as the constraints become clearer over time, as shown in figure 1.5. After identifying assets and defining security goals, you break down those goals into testable constraints. Then as you implement and test those constraints, you may identify new assets to be protected. For example, after implementing your login system, you may give each user a unique tempo-

rary session cookie. This session cookie is itself a new asset that should be protected. Session cookies are discussed in chapter 4.

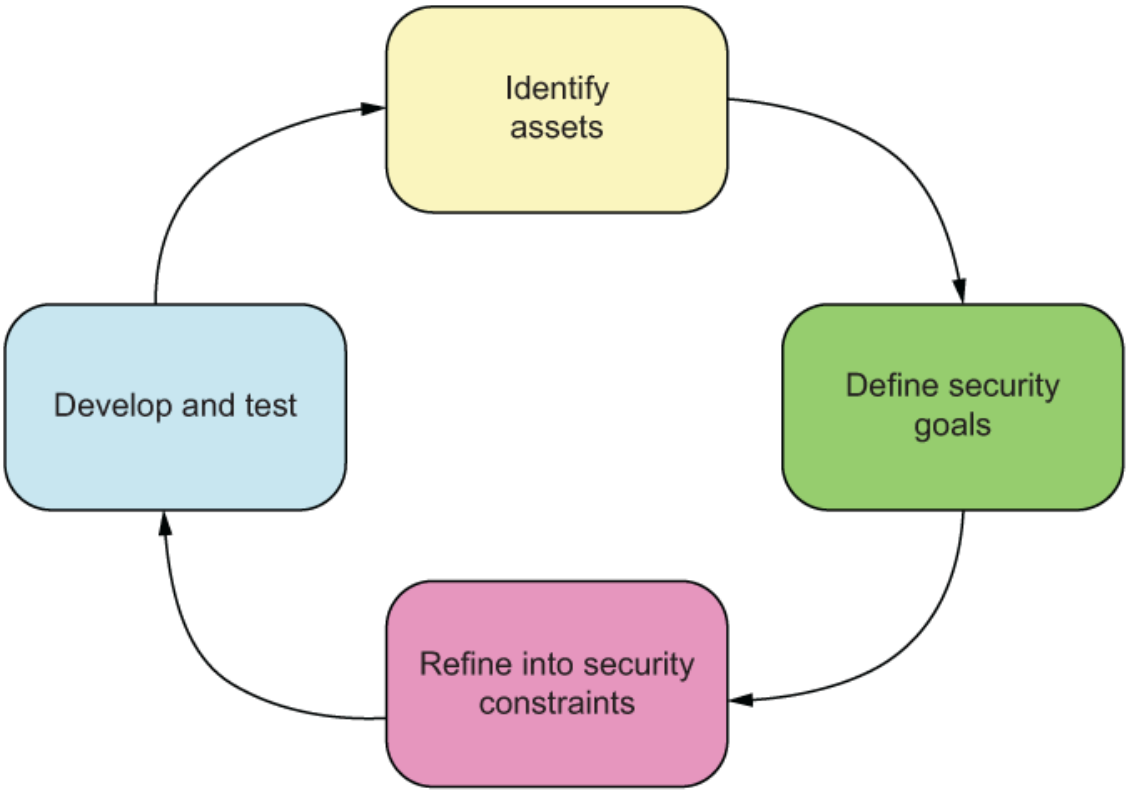

Figure 1.5 Defining security for your API consists of a four-step iterative process of identifying assets, defining the security goals that you need to preserve for those assets, and then breaking those down into testable implementation constraints. Implementation may then identify new assets or goals and so the process continues.

This iterative process shows that security is not a one-off process that can be signed off once and then forgotten about. Just as you wouldn't test the performance of an API only once, you should revisit security goals and assumptions regularly to make sure they are still valid.

1.4.3 Environments and threat models

A good definition of API security must also consider the environment in which your API is to operate and the potential threats that will exist in that environment. A threat is simply any way that a security goal might be violated with respect to one or more of your assets. In a perfect world, you would be able to design an API that achieved its security goals against any threat. But the world is not perfect, and it is rarely possible or economical to prevent all attacks. In some environments some threats are just not worth worrying about. For example, an API for recording race times for a local cycling club probably doesn't need to worry about the at-

tentions of a nation-state intelligence agency, although it may want to prevent riders trying to “improve” their own best times or alter those of other cyclists. By considering realistic threats to your API you can decide where to concentrate your efforts and identify gaps in your defenses.

DEFINITION A threat is an event or set of circumstances that defeats the security goals of your API. For example, an attacker stealing names and address details from your customer database is a threat to confidentiality.

The set of threats that you consider relevant to your API is known as your threat model, and the process of identifying them is known as threat modeling.

DEFINITION Threat modeling is the process of systematically identifying threats to a software system so that they can be recorded, tracked, and mitigated.

There is a famous quote attributed to Dwight D. Eisenhower:

Plans are worthless, but planning is everything.

It is often like that with threat modeling. It is less important exactly how you do threat modeling or where you record the results. What matters is that you do it, because the process of thinking about threats and weaknesses in your system will almost always improve the security of the API.

There are many ways to do threat modeling, but the general process is as follows:

1. Draw a system diagram showing the main logical components of your API.
2. Identify trust boundaries between parts of the system. Everything within a trust boundary is controlled and managed by the same owner, such as a private datacenter or a set of processes running under a single operating system user.
3. Draw arrows to show how data flows between the various parts of the system.

4. Examine each component and data flow in the system and try to identify threats that might undermine your security goals in each case. Pay particular attention to flows that cross trust boundaries. (See the next section for how to do this.)
5. Record threats to ensure they are tracked and managed.

The diagram produced in steps one to three is known as a dataflow diagram, and an example for a fictitious pizza ordering API is given in figure 1.6. The API is accessed by a web application running in a web browser, and also by a native mobile phone app, so these are both drawn as processes in their own trust boundaries. The API server runs in the same datacenter as the database, but they run as different operating system accounts so you can draw further trust boundaries to make this clear. Note that the operating system account boundaries are nested inside the datacenter trust boundary. For the database, I've drawn the database management system (DBMS) process separately from the actual data files. It's often useful to consider threats from users that have direct access to files separately from threats that access the DBMS API because these can be quite different.

IDENTIFYING THREATS

If you pay attention to cybersecurity news stories, it can sometimes seem that there are a bewildering variety of attacks that you need to defend against. While this is partly true, many attacks fall into a few known categories. Several methodologies have been developed to try to systematically identify threats to software systems, and we can use these to identify the kinds of threats that might befall your API. The goal of threat modeling is to identify these general threats, not to enumerate every possible attack.

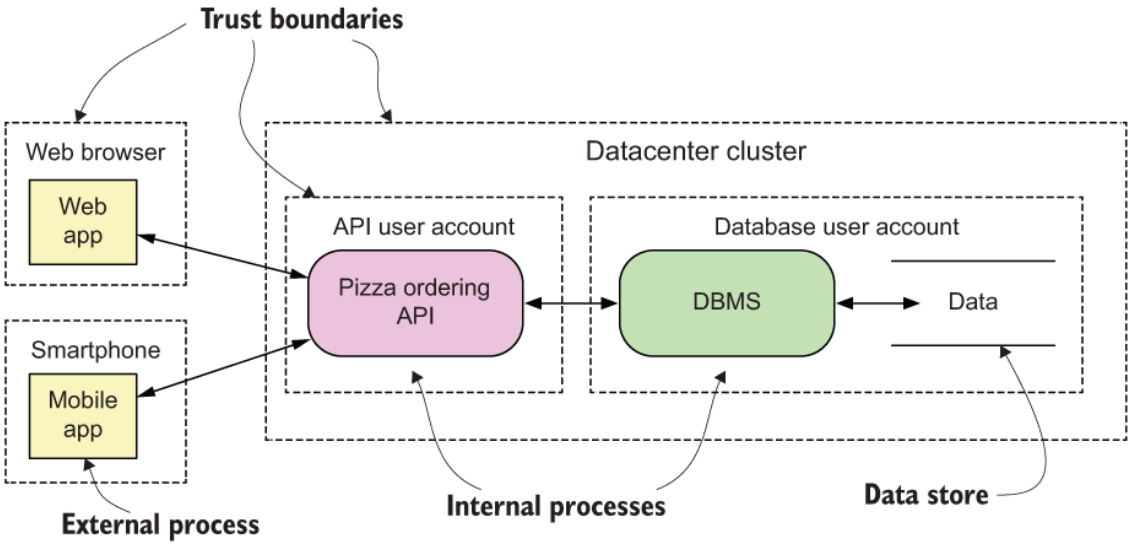

Figure 1.6 An example dataflow diagram, showing processes, data stores and the flow of data between them. Trust boundaries are marked with dashed lines. Internal processes are marked with rounded rectangles, while external entities use squared ends. Note that we include both the database management system (DBMS) process and its data files as separate entities.

One very popular methodology is known by the acronym STRIDE, which stands for:

- Spoofing--Pretending to be somebody else
- Tampering--Altering data, messages, or settings you're not supposed to alter
- Repudiation--Denying that you did something that you really did do
- Information disclosure--Revealing information that should be kept private
- Denial of service--Preventing others from accessing information and services
- Elevation of privilege--Gaining access to functionality you're not supposed to have access to

Each initial in the STRIDE acronym represents a class of threat to your API. General security mechanisms can effectively address each class of threat. For example, spoofing threats, in which somebody pretends to be somebody else, can be addressed by requiring all users to authenticate. Many common threats to API security can be eliminated entirely (or at least significantly mitigated) by the consistent application of a few basic security mechanisms, as you'll see in chapter 3 and the rest of this book.

LEARN ABOUT IT You can learn more about STRIDE, and how to identify specific threats to your applications, through one of many good books about threat modeling. I recommend Adam Shostack's Threat Modeling: Designing for Security (Wiley, 2014) as a good introduction to the subject.

Pop quiz

3. What do the initials CIA stand for when talking about security goals?
4. Which one of the following data flows should you pay the most attention to when threat modeling?
    1. Data flows within a web browser
    2. Data flows that cross trust boundaries
    3. Data flows between internal processes
    4. Data flows between external processes
    5. Data flows between a database and its data files
5. Imagine the following scenario: a rogue system administrator turns off audit logging before performing actions using an API. Which of the STRIDE threats are being abused in this case? Recall from section 1.1 that an audit log records who did what on the system.

The answers are at the end of the chapter.

1.5 Security mechanisms

Threats can be countered by applying security mechanisms that ensure that particular security goals are met. In this section we will run through the most common security mechanisms that you will generally find in every well-designed API:

- Encryption ensures that data can't be read by unauthorized parties, either when it is being transmitted from the API to a client or at rest in a database or filesystem. Modern encryption also ensures that data can't be modified by an attacker.
- Authentication is the process of ensuring that your users and clients are who they say they are.
- Access control (also known as authorization) is the process of ensuring that every request made to your API is appropriately authorized.
- Audit logging is used to ensure that all operations are recorded to allow accountability and proper monitoring of the API.

- Rate-limiting is used to prevent any one user (or group of users) using all of the resources and preventing access for legitimate users.

Figure 1.7 shows how these five processes are typically layered as a series of filters that a request passes through before it is processed by the core logic of your API. As discussed in section 1.3.1, each of these five stages can sometimes be outsourced to an external component such as an API gateway. In this book, you will build each of them from scratch so that you can assess when an external component may be an appropriate choice.

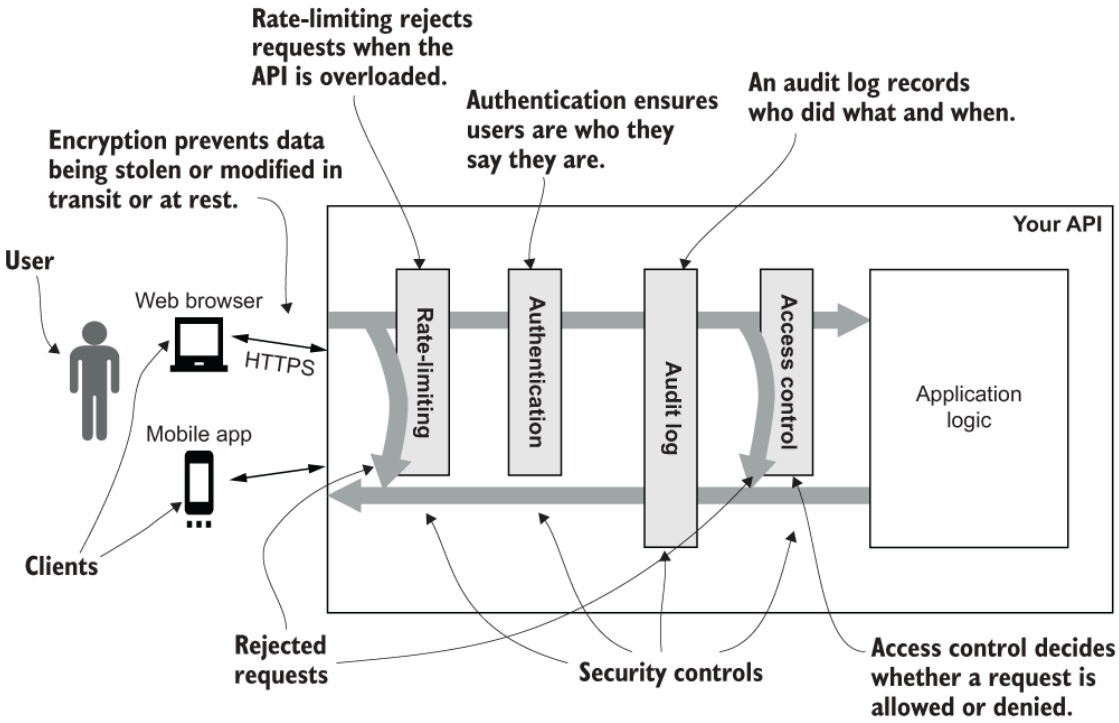

Figure 1.7 When processing a request, a secure API will apply some standard steps. Requests and responses are encrypted using the HTTPS protocol. Rate-limiting is applied to prevent DoS attacks. Then users and clients are identified and authenticated, and a record is made of the access attempt in an access or audit log. Finally, checks are made to decide if this user should be able to perform this request. The outcome of the request should also be recorded in the audit log.

1.5.1 Encryption

The other security mechanisms discussed in this section deal with protecting access to data through the API itself. Encryption is used to protect data when it is outside your API. There are two main cases in which data may be at risk:

- Requests and responses to an API may be at risk as they travel over networks, such as the internet. Encrypting data in transit is used to protect against these threats.
- Data may be at risk from people with access to the disk storage that is used for persistence. Encrypting data at rest is used to protect against these threats.

TLS should be used to encrypt data in transit and is covered in chapter 3. Alternatives to TLS for constrained devices are discussed in chapter 12. Encrypting data at rest is a complex topic with many aspects to consider and is largely beyond the scope of this book. Some considerations for database encryption are discussed in chapter 5.

1.5.2 Identification and authentication

Authentication is the process of verifying whether a user is who they say they are. We are normally concerned with identifying who that user is, but in many cases the easiest way to do that is to have the client tell us who they are and check that they are telling the truth.

The driving test story at the beginning of the chapter illustrates the difference between identification and authentication. When you saw your old friend Alice in the park, you immediately knew who she was due to a shared history of previous interactions. It would be downright bizarre (not to mention rude) if you asked old friends for formal identification! On the other hand, when you attended your driving test it was not surprising that the examiner asked to see your driving license. The examiner has probably never met you before, and a driving test is a situation in which somebody might reasonably lie about who they are, for example, to get a more experienced driver to take the test for them. The driving license authenticates your claim that you are a particular person, and the examiner trusts it because it is issued by an official body and is difficult to fake.

Why do we need to identify the users of an API in the first place? You should always ask this question of any security mechanism you are adding to your API, and the answer should be in terms of one or more of the security goals that you are trying to achieve. You may want to identify users for several reasons:

- You want to record which users performed what actions to ensure accountability.
- You may need to know who a user is to decide what they can do, to enforce confidentiality and integrity goals.
- You may want to only process authenticated requests to avoid anonymous DoS attacks that compromise availability.

Because authentication is the most common method of identifying a user, it is common to talk of "authenticating a user" as a shorthand for identifying that user via authentication. In reality, we never "authenticate" a user themselves but rather claims about their identity such as their username. To authenticate a claim simply means to determine if it is authentic, or genuine. This is usually achieved by asking the user to present some kind of credentials that prove that the claims are correct (they provide credence to the claims, which is where the word "credential" comes from), such as providing a password along with the username that only that user would know.

AUTHENTICATION FACTORS

There are many ways of authenticating a user, which can be divided into three broad categories known as authentication factors :

- Something you know, such as a secret password
- Something you have, like a key or physical device
- Something you are. This refers to biometric factors, such as your unique fingerprint or iris pattern.

Any individual factor of authentication may be compromised. People choose weak passwords or write them down on notes attached to their computer screen, and they mislay physical devices. Although biometric factors can be appealing, they often have high error rates. For this reason, the most secure authentication systems require two or more different factors. For example, your bank may require you to enter a password and then use a device with your bank card to generate a unique login code. This is known as two-factor authentication (2FA) or multi-factor authentication (MFA).

DEFINITION Two-factor authentication (2FA) or multi-factor authentication (MFA) require a user to authenticate with two or more different fac-

tors so that a compromise of any one factor is not enough to grant access to a system.

Note that an authentication factor is different from a credential. Authenticating with two different passwords would still be considered a single factor, because they are both based on something you know. On the other hand, authenticating with a password and a time-based code generated by an app on your phone counts as 2FA because the app on your phone is something you have. Without the app (and the secret key stored inside it), you would not be able to generate the codes.

1.5.3 Access control and authorization

In order to preserve confidentiality and integrity of your assets, it is usually necessary to control who has access to what and what actions they are allowed to perform. For example, a messaging API may want to enforce that users are only allowed to read their own messages and not those of anybody else, or that they can only send messages to users in their friendship group.

NOTE In this book I've used the terms authorization and access control interchangeably, because this is how they are often used in practice. Some authors use the term access control to refer to an overall process including authentication, authorization, and audit logging, or AAA for short.

There are two primary approaches to access control that are used for APIs:

- Identity-based access control first identifies the user and then determines what they can do based on who they are. A user can try to access any resource but may be denied access based on access control rules.
- Capability-based access control uses special tokens or keys known as capabilities to access an API. The capability itself says what operations the bearer can perform rather than who the user is. A capability both names a resource and describes the permissions on it, so a user is not able to access any resource that they do not have a capability for.

Chapters 8 and 9 cover these two approaches to access control in detail.

Capability-based security

The predominant approach to access control is identity-based, where who you are determines what you can do. When you run an application on your computer, it runs with the same permissions that you have. It can read and write all the files that you can read and write and perform all the same actions that you can do. In a capability-based system, permissions are based on unforgeable references known as capabilities (or keys). A user or an application can only read a file if they hold a capability that allows them to read that specific file. This is a bit like a physical key that you use in the real world; whoever holds the key can open the door that it unlocks. Just like a real key typically only unlocks a single door, capabilities are typically also restricted to just one object or file. A user may need many capabilities to get their work done, and capability systems provide mechanisms for managing all these capabilities in a user-friendly way. Capability-based access control is covered in detail in chapter 9.

It is even possible to design applications and their APIs to not need any access control at all. A wiki is a type of website invented by Ward Cunningham, where users collaborate to author articles about some topic or topics. The most famous wiki is Wikipedia, the online encyclopedia that is one of the most viewed sites on the web. A wiki is unusual in that it has no access controls at all. Any user can view and edit any page, and even create new pages. Instead of access controls, a wiki provides extensive version control capabilities so that malicious edits can be easily undone. An audit log of edits provides accountability because it is easy to see who changed what and to revert those changes if necessary. Social norms develop to discourage antisocial behavior. Even so, large wikis like Wikipedia often have some explicit access control policies so that articles can be locked temporarily to prevent “edit wars” when two users disagree strongly or in cases of persistent vandalism.

1.5.4 Audit logging

An audit log is a record of every operation performed using your API. The purpose of an audit log is to ensure accountability. It can be used after a security breach as part of a forensic investigation to find out what went wrong, but also analyzed in real-time by log analysis tools to identity at-

tacks in progress and other suspicious behavior. A good audit log can be used to answer the following kinds of questions:

- Who performed the action and what client did they use?
- When was the request received?
- What kind of request was it, such as a read or modify operation?
- What resource was being accessed?
- Was the request successful? If not, why?
- What other requests did they make around the same time?

It's essential that audit logs are protected from tampering, and they often contain personally identifiable information that should be kept confidential. You'll learn more about audit logging in chapter 3.

DEFINITION Personally identifiable information, or PII, is any information that relates to an individual person and can help to identify that person. For example, their name or address, or their date and place of birth. Many countries have data protection laws like the GDPR, which strictly control how PII may be stored and used.

1.5.5 Rate-limiting

The last mechanisms we will consider are for preserving availability in the face of malicious or accidental DoS attacks. A DoS attack works by exhausting some finite resource that your API requires to service legitimate requests. Such resources include CPU time, memory and disk usage, power, and so on. By flooding your API with bogus requests, these resources become tied up servicing those requests and not others. As well as sending large numbers of requests, an attacker may also send overly large requests that consume a lot of memory or send requests very slowly so that resources are tied up for a long time without the malicious client needing to expend much effort.

The key to fending off these attacks is to recognize that a client (or group of clients) is using more than their fair share of some resource: time, memory, number of connections, and so on. By limiting the resources that any one user is allowed to consume, we can reduce the risk of attack. Once a user has authenticated, your application can enforce quotas that restrict what they are allowed to do. For example, you might restrict each user to a certain number of API requests per hour, preventing them from

flooding the system with too many requests. There are often business reasons to do this for billing purposes, as well as security benefits. Due to the application-specific nature of quotas, we won't cover them further in this book.

DEFINITION A quota is a limit on the number of resources that an individual user account can consume. For example, you may only allow a user to post five messages per day.

Before a user has logged in you can apply simpler rate-limiting to restrict the number of requests overall, or from a particular IP address or range. To apply rate-limiting, the API (or a load balancer) keeps track of how many requests per second it is serving. Once a predefined limit is reached then the system rejects new requests until the rate falls back under the limit. A rate-limiter can either completely close connections when the limit is exceeded or else slow down the processing of requests, a process known as throttling. When a distributed DoS is in progress, malicious requests will be coming from many different machines on different IP addresses. It is therefore important to be able to apply rate-limiting to a whole group of clients rather than individually. Rate-limiting attempts to ensure that large floods of requests are rejected before the system is completely overwhelmed and ceases functioning entirely.

DEFINITION Throttling is a process by which a client's requests are slowed down without disconnecting the client completely. Throttling can be achieved either by queueing requests for later processing, or else by responding to the requests with a status code telling the client to slow down. If the client doesn't slow down, then subsequent requests are rejected.

The most important aspect of rate-limiting is that it should use fewer resources than would be used if the request were processed normally. For this reason, rate-limiting is often performed in highly optimized code running in an off-the-shelf load balancer, reverse proxy, or API gateway that can sit in front of your API to protect it from DoS attacks rather than having to add this code to each API. Some commercial companies offer DoS protection as a service. These companies have large global infrastructure that is able to absorb the traffic from a DoS attack and quickly block abusive clients.

In the next chapter, we will get our hands dirty with a real API and apply some of the techniques we have discussed in this chapter.

Pop quiz

6. Which of the STRIDE threats does rate-limiting protect against?
    1. Spoofing
    2. Tampering
    3. Repudiation
    4. Information disclosure
    5. Denial of service
    6. Elevation of privilege
7. The WebAuthn standard (**https://www.w3.org/TR/webauthn/**) allows hardware security keys to be used by a user to authenticate to a website. Which of the three authentication factors from section 1.5.1 best describes this method of authentication?

The answers are at the end of the chapter.

Answers to pop quiz questions

1. c, e, and f. While other aspects of security may be relevant to different APIs, these three disciplines are the bedrock of API security.
2. d. An API gateway is a specialized type of reverse proxy.
3. Confidentiality, Integrity, and Availability.
4. b. Data flows that cross trust boundaries are the most likely place for threats to occur. APIs often exist at trust boundaries.
5. Repudiation. By disabling audit logging, the rogue system administrator will later be able to deny performing actions on the system as there will be no record.
6. e. Rate-limiting primarily protects against denial of service attacks by preventing a single attacker from overloading the API with requests.
7. A hardware security key is something you have. They are usually small devices that can be plugged into a USB port on your laptop and can be attached to your key ring.

Summary

- You learned what an API is and the elements of API security, drawing on aspects of information security, network security, and application security.
- You can define security for your API in terms of assets and security goals.
- The basic API security goals are confidentiality, integrity, and availability, as well as accountability, privacy, and others.
- You can identify threats and assess risk using frameworks such as STRIDE.
- Security mechanisms can be used to achieve your security goals, including encryption, authentication, access control, audit logging, and rate-limiting.

---

1. In this context, the newer term TLS is rarely used.