2

HOW THE INTERNET WORKS

To become an expert on web security, you need a firm grasp of the internet's underlying web technologies and protocols. This chapter examines the Internet Protocol Suite, which dictates how computers exchange data over the web. You'll also learn about stateful connections and encryption, which are key elements of the modern web. I'll highlight where security holes tend to appear along the way.

The Internet Protocol Suite

In the early days of the internet, data exchange wasn't reliable. The first message sent over the *Advanced Research Projects Agency Network (ARPANET)*, the predecessor to the internet, was a `LOGIN` command destined for a remote computer at Stanford University. The network sent the first two letters, `LO`, and then crashed. This was a problem for the US military, which was

looking for a way to connect remote computers so that they could continue to exchange information even if a Soviet nuclear strike took various parts of the network offline.

To address this problem, the network engineers developed the *Transmission Control Protocol (TCP)* to ensure a reliable exchange of information between computers. TCP is one of about 20 network protocols that are collectively referred to as the *internet protocol suite*. When a computer sends a message to another machine via TCP, the message is split into data packets that are sent toward their eventual destination with a destination address. The computers that make up the internet push each packet toward the destination without having to process the whole message.

Once the recipient computer receives the packets, it assembles them back into a usable order according to the *sequence number* on each packet. Every time the recipient receives a packet, it sends a receipt. If the recipient fails to acknowledge receipt of a packet, the sender resends that packet, possibly along a different network path. In this way, TCP allows computers to deliver data across a network that is expected to be unreliable.

TCP has undergone significant improvements as the internet has grown. Packets are now sent with a *checksum* that allows recipients to detect data corruption and determine whether packets need to be resent. Senders also preemptively adjust the rate at which they send data according to how fast it's being consumed. (Internet servers are usually magnitudes more powerful than the clients that receive their messages, so they need to be careful not to overwhelm the client's capacity.)

NOTE *TCP remains the most common protocol because of its delivery guarantees, but nowadays, several other protocols are also used over the internet. The* User Datagram Protocol (UDP)*, for instance, is a newer protocol that deliberately allows packets to be dropped so that data can be streamed at a constant rate. UDP is commonly used for streaming live video, since consumers prefer a few dropped frames over having their feed delayed when the network gets congested.*

Internet Protocol Addresses

Data packets on the internet are sent to *Internet Protocol (IP) addresses*, numbers assigned to individual internet-connected computers. Each IP address must be unique, so new IP addresses are issued in a structured fashion.

At the highest level, the *Internet Corporation for Assigned Names and Numbers (ICANN)* allots blocks of IP addresses to regional authorities. These regional authorities then grant the blocks of addresses to *internet service providers (ISPs)* and hosting companies within their region. When you connect your browser to the internet, your ISP assigns your computer an IP address that stays fixed for a few months. (ISPs tend to rotate IP addresses for clients periodically.) Similarly, companies that host content on the internet are assigned an IP address for each server they connect to the network.

IP addresses are binary numbers, generally written in *IP version 4 (IPv4)* syntax, which allows for 2^{32} (4,294,967,296) addresses. Google's

domain name server, for instance, has the address `8.8.8.8`. Because IPv4 addresses are getting used up at a rate that isn't sustainable, the internet is shifting to *IP version 6 (IPv6)* addresses to allow for more connected devices, represented as eight groups of four hexadecimal digits separated by colons (for example: `2001:0db8:0000:0042:0000:8a2e:0370:7334`).

The Domain Name System

Browsers and other internet-connected software can recognize and route traffic to IP addresses, but IP addresses aren't particularly memorable for humans. To make website addresses friendlier to users, we use a global directory called the *Domain Name System (DNS)* to translate human-readable *domains* like *example.com* to IP addresses like `93.184.216.119`. Domain names are simply placeholders for IP addresses. Domain names, like IP addresses, are unique, and have to be registered before use with private organizations called *domain registrars*.

When browsers encounter a domain name for the first time, they use a local *domain name server* (typically hosted by an ISP) to look it up, and then cache the result to prevent time-consuming lookups in the future. This caching behavior means that new domains or changes to existing domains take a while to propagate on the internet. Exactly how long this propagation takes is controlled by the *time-to-live (TTL)* variable, which is set on the DNS record and instructs DNS caches when to expire the record. DNS caching enables a type of attack called *DNS poisoning*, whereby a local DNS cache is deliberately corrupted so that data is routed to a server controlled by an attacker.

In addition to returning IP addresses for particular domains, domain name servers host records that can describe domain aliases via *canonical name (CNAME) records* that allow multiple domain names to point to the same IP address. DNS can also help route email by using *mail exchange (MX)* records. We'll examine how DNS records can help combat unsolicited email (spam) in Chapter 16.

Application Layer Protocols

The technical standards used for communication on the internet, including TCP, are defined and developed by the *Internet Engineering Task Force (IETF)*. These standards are called the *internet protocol suite*. TCP allows two computers to reliably exchange data on the internet, but it doesn't dictate how the data being sent should be interpreted. For that to happen, both computers need to agree to exchange information through another, higher-level protocol in the suite. Protocols that build on top of TCP (or UDP) are called *application layer protocols*. Figure 2-1 illustrates how application layer protocols sit above TCP in the internet protocol suite.

The lower-level protocols of the internet protocol suite provide basic data routing over a network, while the higher-level protocols in the application layer provide more structure for applications exchanging data. Many types of applications use TCP as a transport mechanism on the internet.

For example, emails are sent using the Simple Mail Transport Protocol (SMTP), instant messaging software often uses the Extensible Messaging and Presence Protocol (XMPP), file servers make downloads available via the File Transfer Protocol (FTP), and web servers use the HyperText Transfer Protocol (HTTP). Because the web is our chief focus, let's look at HTTP in more detail.

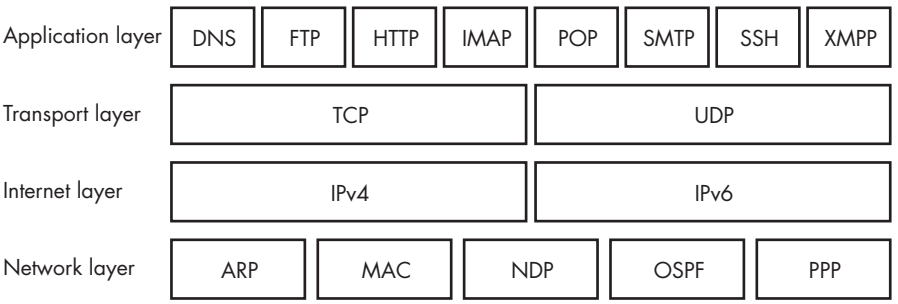

Figure 2-1: The various layers that make up the internet protocol suite

HyperText Transfer Protocol

Web servers use the *HyperText Transfer Protocol (HTTP)* to transport web pages and their resources to *user agents* such as web browsers. In an HTTP conversation, the user agent generates *requests* for particular resources. Web servers, expecting these requests, return *responses* containing either the requested resource, or an error code if the request can't be fulfilled. Both HTTP requests and responses are plaintext messages, though they're often sent in compressed and encrypted form. All of the exploits described in this book use HTTP in some fashion, so it's worth knowing how the requests and responses that make up HTTP conversations work in detail.

HTTP Requests

An HTTP request sent by a browser consists of the following elements:

Method Also known as a *verb*, this describes the action that the user agent wants the server to perform.

Universal resource locator (URL) This describes the resource being manipulated or fetched.

Headers These supply metadata such as the type of content the user agent is expecting or whether it accepts compressed responses.

Body This optional component contains any extra data that needs to be sent to the server.

Listing 2-1 shows an HTTP request.

```
GET❶ http://example.com/❷
❸ User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.99 Safari/537.36
❹ Accept: text/html,application/xhtml+xml,application/xml; */*
```

```
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
```

Listing 2-1: A simple HTTP request

The method ❶ and the URL ❷ appear on the first line. These are followed by HTTP headers on separate lines. The `User-Agent` header ❸ tells the website the type of browser that is making the request. The `Accept` header ❹ tells the website the type of content the browser is expecting.

`GET` requests are the most common type of request on the internet—they request a particular resource on the web server, identified by a specific URL. The response to a `GET` request will contain a resource: perhaps a web page, an image, or even the results of a search request. The example request in Listing 2-1 represents an attempt to load the home page of *example.com*, and would be generated when a user types *example.com* in the browser's navigation bar.

If the browser needs to send information to the server, rather than just fetch data, it typically uses a `POST` request. When you fill out a form on a web page and submit it, the browser sends a `POST` request. Because `POST` requests contain information sent to the server, the browser sends that information in a *request body*, after the HTTP headers.

In Chapter 8, you'll see why it's important to use `POST` rather than `GET` requests when sending data to your server. Websites that erroneously use `GET` requests for doing anything other than retrieving resources are vulnerable to cross-site request forgery attacks.

When writing a website, you may also encounter `PUT`, `PATCH`, and `DELETE` requests. These are used to upload, edit, or delete resources on the server, respectively, and are typically triggered by JavaScript embedded in a web page. Table 2-1 documents a handful of other methods that are worth knowing about.

Table 2-1: The Lesser-Known HTTP Methods

HTTP method	Function and implementation
`HEAD`	A `HEAD` request retrieves the same information as a `GET` request, but instructs the server to return the response without a body (in other words, the useful part). If you implement a `GET` method on your web server, the server will generally respond to `HEAD` requests automatically.
`CONNECT`	`CONNECT` initiates two-way communications. You'll use it in your HTTP client code if you ever have to connect through a proxy.
`OPTIONS`	Sending an `OPTIONS` request lets a user agent ask what other methods are supported by a resource. Your web server will generally respond to `OPTIONS` requests by inferring which other methods you have implemented.
`TRACE`	A response to a `TRACE` request will contain an exact copy of the original HTTP request, so the client can see what (if any) alterations were made by intermediate servers. This sounds useful, but it's generally recommended that you turn off `TRACE` requests in your web server, because they can act as a security hole. For instance, they can allow malicious JavaScript injected into a page to access cookies that have been deliberately made inaccessible to JavaScript.

Once a web server receives an HTTP request, it replies to the user agent with an HTTP response. Let's break down how responses are structured.

HTTP Responses

HTTP responses sent back by a web server begin with a protocol description, a three-digit *status code*, and, typically, a *status message* that indicates whether the request can be fulfilled. The response also contains headers providing metadata that instructs the browser how to treat the content. Finally, most responses contain a body that itself contains the requested resource. Listing 2-2 shows the contents of a simple HTTP response.

```
HTTP/1.1❶ 200❷ OK❸
❹ Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Content-Length: 606

❺ <!doctype html>
<html>
   <head>
      <title>Example Domain</title>
    ❻ <style type="text/css">
         body {
            background-color: #f0f0f2;
            font-family: "Open Sans", "Helvetica Neue", Helvetica, sans-serif;
         }
         div {
            width: 600px;
            padding: 50px;
            background-color: #fff;
            border-radius: 1em;
         }
      </style>
   </head>
 ❼ <body>
      <div>
         <h1>Example Domain</h1>
         <p>This domain is established to be used for illustrative examples.</p>
         <p>
           <a href="http://www.iana.org/domains/example">More information...</a>
         </p>
      </div>
   </body>
</html>
```

Listing 2-2: An HTTP response from example.com*, the world's least interesting website*

The response begins with the protocol description ❶, the status code ❷, and the status message ❸. Status codes formatted as 2*xx* indicate that the request was understood, accepted, and responded to. Codes formatted as

`3xx` redirect the client to a different URL. Codes formatted as `4xx` indicate a client error: the browser generated an apparently invalid request. (The most common error of this type is `HTTP 404 Not Found`). Codes formatted as `5xx` indicate a server error: the request was valid, but the server was unable to fulfill the request.

Next are the HTTP headers ❹. Almost all HTTP responses include a `Content-Type` header that indicates the kind of data being returned. Responses to `GET` requests also often contain a `Cache-Control` header to indicate that the client should cache large resources (for example, images) locally.

If the HTTP response is successful, the body contains the resource the client was trying to access as well as *HyperText Markup Language (HTML)* ❺ describing the structure of the requested web page. In this case, the response contains styling information ❻ as well as the page content itself ❼. Other types of responses may return JavaScript code, Cascading Style Sheets (CSS) used for styling HTML, or binary data in the body.

Stateful Connections

Web servers typically deal with many user agents at once, but HTTP does nothing to distinguish which requests are coming from which user agent. This wasn't an important consideration in the early days of the internet, because web pages were largely read-only. Modern websites, however, often allow users to log in and will track their activity as they visit and interact with different pages. To allow for this, HTTP conversations need to be made stateful. A connection or conversation between a client and a server is *stateful* when they perform a "handshake" and continue to send packets back and forth until one of the communicating parties decides to terminate the connection.

When a web server wants to keep track of which user it's responding to with each request, and thus achieve a stateful HTTP conversation, it needs to establish a mechanism to track the user agent as it makes the subsequent requests. The entire conversation between a particular user agent and a web server is called an *HTTP session*. The most common way of tracking sessions is for the server to send back a `Set-Cookie` header in the initial HTTP response. This asks the user agent receiving the response to store a *cookie*, a small snippet of text data pertaining to that particular web domain. The user agent then returns the same data in the `Cookie` header of any subsequent HTTP request to the web server. If implemented correctly, the contents of the cookie being passed back and forth uniquely identify the user agent and hence establish the HTTP session.

Session information contained in cookies is a juicy target for hackers. If an attacker steals another user's cookie, they can pretend to be that user on the website. Similarly, if an attacker successfully persuades a website to accept a forged cookie, they can impersonate any user they please. We'll look at various methods of stealing and forging cookies later in the book.

Encryption

When the web was first invented, HTTP requests and responses were sent in plaintext form, which meant they could be read by anyone intercepting the data packets; this kind of interception is known as a *man-in-the-middle attack*. Because private communication and online transactions are common on the modern web, web servers and browsers protect their users from such attacks by using *encryption*, a method of disguising the contents of messages from prying eyes by encoding them during transmission.

To secure their communications, web servers and browsers send requests and responses by using *Transport Layer Security (TLS)*, a method of encryption that provides both privacy and data integrity. TLS ensures that packets intercepted by a third party can't be decrypted without the appropriate encryption keys. It also ensures that any attempt to tamper with the packets will be detectable, which ensures data integrity.

HTTP conversations conducted using TLS are called *HTTP Secure (HTTPS)*. HTTPS requires the client and server to perform a *TLS handshake* in which both parties agree on an encryption method (a cipher) and exchange encryption keys. Once the handshake is complete, any further messages (both requests and responses) will be opaque to outsiders.

Encryption is a complex topic but is key to securing your website. We'll examine how to enable encryption for your website in Chapter 13.

Summary

In this chapter, you learned about the plumbing of the internet. TCP enables reliable communication between internet-connected computers that each have an IP address. The Domain Name System provides human-readable aliases for IP addresses. HTTP builds on top of TCP to send HTTP requests from user agents (such as web browsers) to web servers, which in turn reply with HTTP responses. Each request is sent to a specific URL, and you learned about various types of HTTP methods. Web servers respond with status codes, and send back cookies to initiate stateful connections. Finally, encryption (in the form of HTTPS) can be used to secure communication between a user agent and a web server.

In the next chapter, you'll take a look at what happens when a web browser receives an HTTP response—how a web page is rendered, and how user actions can generate more HTTP requests.