Tangled Je Web

A Guide to Securing Modern Web Applications

Michal Zalewski

no starch Press

THE TANGLED WEB

A Guide to Securing Modern Web Applications

by Michal Zalewski

San Francisco

15 14 13 12 11 1 2 3 4 5 6 7 8 9
ISBN-10: 1-59327-388-6 ISBN-13: 978-1-59327-388-0
Publisher: William Pollock Production Editor: Serena Yang Cover Illustration: Hugh D'Andrade Interior Design: Octopod Studios Developmental Editor: William Pollock Technical Reviewer: Chris Evans Copyeditor: Paula L. Fleming Compositor: Serena Yang Proofreader: Ward Webber Indexer: Nancy Guenther
For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc. 38 Ringold Street, San Francisco, CA 94103 phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com
Library of Congress Cataloging-in-Publication Data
Zalewski, Michal.
The tangled Web : a guide to securing modern Web applications / Michal Zalewski.
p. cm. Includes bibliographical references and index. ISBN-13: 978-1-59327-388-0 (pbk.) ISBN-10: 1-59327-388-6 (pbk.) 1. Computer networks--Security measures. 2. Browsers (Computer programs) 3. Computer security. I. Title. TK5105.59.Z354 2011 005.8--dc23
2011039636

For my son

BRIEF CONTENTS

CONTENTS IN DETAIL

PREFACE

Just fifteen years ago, the Web was as simple as it was unimportant: a quirky mechanism that allowed a handful of students, plus a bunch of asocial, basement-dwelling geeks, to visit each other's home pages dedi- cated to science, pets, or poetry. Today, it is the platform of choice for writing complex, interactive applications (from mail clients to image editors to computer games) and a medium reaching hundreds of millions of casual users around the globe. It is also an essential tool of commerce, important enough to be credited for caus- ing a recession when the 1999 to 2001 dot-com bubble burst.

This progression from obscurity to ubiquity was amazingly fast, even by the standards we are accustomed to in today's information age—and its speed of ascent brought with it an unexpected problem. The design flaws

and implementation shortcomings of the World Wide Web are those of a technology that never aspired to its current status and never had a chance to pause and look back at previous mistakes. The resulting issues have quickly emerged as some of the most significant and prevalent threats to data secu- rity today: As it turns out, the protocol design standards one would apply to a black-on-gray home page full of dancing hamsters are not necessarily the same for an online shop that processes millions of credit card transactions every year.

When taking a look at the past decade, it is difficult not to be slightly disappointed: Nearly every single noteworthy online application devised so far has had to pay a price for the corners cut in the early days of the Web. Heck, *xssed.com*, a site dedicated to tracking a narrow subset of web-related security glitches, amassed some 50,000 entries in about three years of opera- tion. Yet, browser vendors are largely unfazed, and the security community itself has offered little insight or advice on how to cope with the widespread misery. Instead, many security experts stick to building byzantine vulnerabil- ity taxonomies and engage in habitual but vague hand wringing about the supposed causes of this mess.

Part of the problem is that said experts have long been dismissive of the whole web security ruckus, unable to understand what it was all about. They have been quick to label web security flaws as trivial manifestations of the *confused deputy problem** or of some other catchy label outlined in a trade jour- nal three decades ago. And why should they care about web security, anyway? What is the impact of an obscene comment injected onto a dull pet-themed home page compared to the gravity of a traditional system-compromise flaw?

In retrospect, I'm pretty sure most of us are biting our tongues. Not only has the Web turned out to matter a lot more than originally expected, but we've failed to pay attention to some fundamental characteristics that put it well outside our comfort zone. After all, even the best-designed and most thoroughly audited web applications have far more issues, far more frequently, than their nonweb counterparts.

We all messed up, and it is time to repent. In the interest of repentance, *The Tangled Web* tries to take a small step toward much-needed normalcy, and as such, it may be the first publication to provide a systematic and thorough analysis of the current state of affairs in the world of web application security. In the process of doing so, it aims to shed light on the uniqueness of the secu- rity challenges that we—security engineers, web developers, and users—have to face every day.

The layout of this book is centered on exploring some of the most prom- inent, high-level browser building blocks and various security-relevant topics derived from this narrative. I have taken this approach because it seems to be more informative and intuitive than simply enumerating the issues using an

[*] *Confused deputy problem* is a generic concept in information security used to refer to a broad class of design or implementation flaws. The term describes any vector that allows the attacker to trick a program into misusing some "authority" (access privileges) to manipulate a resource in an unintended manner—presumably one that is beneficial to the attacker, however that benefit is defined. The phrase "confused deputy" is regularly invoked by security researchers in academia, but since virtually all real-world security problems could be placed in this bucket when considered at some level of abstraction, this term is nearly meaningless.

arbitrarily chosen taxonomy (a practice seen in many other information security books). I hope, too, that this approach will make *The Tangled Web* a better read.

For readers looking for quick answers, I decided to include quick engi- neering cheat sheets at the end of many of the chapters. These cheat sheets outline sensible approaches to some of the most commonly encountered problems in web application design. In addition, the final part of the book offers a quick glossary of the well-known implementation vulnerabilities that one may come across.

Acknowledgments

Many parts of *The Tangled Web* have their roots in the research done for Google's *Browser Security Handbook*, a technical wiki I put together in 2008 and released publicly under a Creative Commons license. You can browse the original document online at *http://code.google.com/p/browsersec/.*

I am fortunate to be with a company that allowed me to pursue this project—and delighted to be working with a number of talented peers who provided excellent input to make the *Browser Security Handbook* more useful and accurate. In particular, thanks to Filipe Almeida, Drew Hintz, Marius Schilder, and Parisa Tabriz for their assistance.

I am also proud to be standing on the shoulders of giants. This book owes a lot to the research on browser security done by members of the informa- tion security community. Special credit goes to Adam Barth, Collin Jackson, Chris Evans, Jesse Ruderman, Billy Rios, and Eduardo Vela Nava for the advancement of our understanding of this field.

Thank you all—and keep up the good work.

SECURITY IN THE WORLD OF WEB APPLICATIONS

To provide proper context for the technical discus- sions later in the book, it seems prudent to first of all explain what the field of security engineering tries to achieve and then to outline why, in this otherwise well- studied context, web applications deserve special treat- ment. So, shall we?

Information Security in a Nutshell

On the face of it, the field of information security appears to be a mature, well-defined, and accomplished branch of computer science. Resident experts eagerly assert the importance of their area of expertise by pointing to large sets of neatly cataloged security flaws, invariably attributed to security-illiterate developers, while their fellow theoreticians note how all these problems would have been prevented by adhering to this year's hottest security methodology.

A commercial industry thrives in the vicinity, offering various nonbinding security assurances to everyone, from casual computer users to giant interna- tional corporations.

Yet, for several decades, we have in essence completely failed to come up with even the most rudimentary usable frameworks for understanding and assessing the security of modern software. Save for several brilliant treatises and limited-scale experiments, we do not even have any real-world success stories to share. The focus is almost exclusively on reactive, secondary secu- rity measures (such as vulnerability management, malware and attack detec- tion, sandboxing, and so forth) and perhaps on selectively pointing out flaws in somebody else's code. The frustrating, jealously guarded secret is that when it comes to enabling others to develop secure systems, we deliver

far less value than should be expected; the modern Web is no exception.
Let’s look at some of the most alluring approaches to ensuring informa- tion security and try to figure out why they have not made a difference so far.

Flirting with Formal Solutions

Perhaps the most obvious tool for building secure programs is to algorithmi- cally prove they behave just the right way. This is a simple premise that intu- itively should be within the realm of possibility—so why hasn’t this approach netted us much?

Well, let’s start with the adjective *secure* itself: What is it supposed to convey, precisely? Security seems like an intuitive concept, but in the world of comput- ing, it escapes all attempts to usefully define it. Sure, we can restate the prob- lem in catchy yet largely unhelpful ways, but you know there’s a problem when one of the definitions most frequently cited by practitioners[*] is this:

A system is secure if it behaves precisely in the manner intended— and does nothing more.

This definition is neat and vaguely outlines an abstract goal, but it tells very little about how to achieve it. It’s computer science, but in terms of spec- ificity, it bears a striking resemblance to a poem by Victor Hugo:

Love is a portion of the soul itself, and it is of the same nature as the celestial breathing of the atmosphere of paradise.

One could argue that practitioners are not the ones to be asked for nuanced definitions, but go ahead and pose the same question to a group of academics and they’ll offer you roughly the same answer. For example, the following common academic definition traces back to the Bell-La Padula secu- rity model, published in the 1960s. (This was one of about a dozen attempts to formalize the requirements for secure systems, in this case in terms of a finite state machine;[1] it is also one of the most notable ones.)

A system is secure if and only if it starts in a secure state and cannot enter an insecure state.

[*] The quote is attributed originally to Ivan Arce, a renowned vulnerability hunter, circa 2000; since then, it has been used by Crispin Cowan, Michael Howard, Anton Chuvakin, and scores of other security experts.

Definitions along these lines are fundamentally true, of course, and may serve as the basis for dissertations or even a couple of government grants. But in practice, models built on these foundations are bound to be nearly useless for generalized, real-world software engineering for at least three reasons:

- **There is no way to define desirable behavior for a sufficiently complex computer system.** No single authority can define what the “intended manner” or “secure states” should be for an operating system or a web browser. The interests of users, system owners, data providers, business process owners, and software and hardware vendors tend to differ sig- nificantly and shift rapidly—when the stakeholders are capable and will- ing to clearly and honestly disclose their interests to begin with. To add insult to injury, sociology and game theory suggest that computing a sim- ple sum of these particular interests may not actually result in a benefi- cial outcome. This dilemma, known as “the tragedy of the commons,” is central to many disputes over the future of the Internet.
- **Wishful thinking does not automatically map to formal constraints.** Even if we can reach a perfect, high-level agreement about how the sys- tem should behave in a subset of cases, it is nearly impossible to formal- ize such expectations as a set of permissible inputs, program states, and state transitions, which is a prerequisite for almost every type of formal analysis. Quite simply, intuitive concepts such as “I do not want my mail to be read by others,” do not translate to mathematical models particu- larly well. Several exotic approaches will allow such vague requirements to be at least partly formalized, but they put heavy constraints on software- engineering processes and often result in rulesets and models that are far more complicated than the validated algorithms themselves. And, in turn, they are likely to need their own correctness to be proven . . . *ad infinitum*.
- **Software behavior is very hard to conclusively analyze.** Static analysis of computer programs with the intent to prove that they will always behave according to a detailed specification is a task that no one has managed to believably demonstrate in complex, real-world scenarios (though, as you might expect, limited success in highly constrained settings or with very narrow goals is possible). Many cases are likely to be impossible to solve in practice (due to computational complexity) and may even turn out to be completely undecidable due to the halting

problem.[*]

Perhaps more frustrating than the vagueness and uselessness of the early definitions is that as the decades have passed, little or no progress has been made toward something better. In fact, an academic paper released in 2001 by the Naval Research Laboratory backtracks on some of the earlier work and arrives at a much more casual, enumerative definition of software security— one that explicitly disclaims its imperfection and incompleteness.[2]

[*] In 1936, Alan Turing showed that (paraphrasing slightly) it is not possible to devise an algorithm that can generally decide the outcome of other algorithms. Naturally, some algorithms are very much decidable by conducting case-specific proofs, just not all of them.

A system is secure if it adequately protects information that it pro- cesses against unauthorized disclosure, unauthorized modification, and unauthorized withholding (also called denial of service). We say "adequately" because no practical system can achieve these goals without qualification; security is inherently relative.

The paper also provides a retrospective assessment of earlier efforts and the unacceptable sacrifices made to preserve the theoretical purity of said models:

Experience has shown that, on one hand, the axioms of the Bell- La Padula model are overly restrictive: they disallow operations that users require in practical applications. On the other hand, trusted subjects, which are the mechanism provided to overcome some of these restrictions, are not restricted enough. . . . Consequently, developers have had to develop ad hoc specifications for the desired behavior of trusted processes in each individual system.

In the end, regardless of the number of elegant, competing models intro- duced, all attempts to understand and evaluate the security of real-world soft- ware using algorithmic foundations seem bound to fail. This leaves developers and security experts with no method to make authoritative, future-looking statements about the quality of produced code. So, what other options are on the table?

Enter Risk Management

In the absence of formal assurances and provable metrics, and given the frightening prevalence of security flaws in key software relied upon by mod- ern societies, businesses flock to another catchy concept: *risk management*. The idea of risk management, applied successfully to the insurance business (with perhaps a bit less success in the financial world), simply states that system owners should learn to live with vulnerabilities that cannot be addressed in a cost-effective way and, in general, should scale efforts accord- ing to the following formula:

risk = probability of an event x *maximum loss*

For example, according to this doctrine, if having some unimportant workstation compromised yearly won't cost the company more than $1,000 in lost productivity, the organization should just budget for this loss and move on, rather than spend say $100,000 on additional security measures or con- tingency and monitoring plans to prevent the loss. According to the doctrine of risk management, the money would be better spent on isolating, securing, and monitoring the mission-critical mainframe that churns out billing records for all customers.

Naturally, it's prudent to prioritize security efforts. The problem is that when risk management is done strictly by the numbers, it does little to help us to understand, contain, and manage real-world problems. Instead, it intro- duces a dangerous fallacy: that structured inadequacy is almost as good as adequacy and that underfunded security efforts *plus* risk management are about as good as properly funded security work.

Guess what? No dice.

- **In interconnected systems, losses are not capped and are not tied to an asset.** Strict risk management depends on the ability to estimate typi- cal and maximum cost associated with the compromise of a resource. Unfortunately, the only way to do this is to overlook the fact that many of the most spectacular security breaches—such as the attacks on TJX[*] or Microsoft[†]—began at relatively unimportant and neglected entry points. These initial intrusions soon escalated and eventually resulted in the nearly complete compromise of critical infrastructure, bypassing any superficial network compartmentalization on their way. In typical by-the-numbers risk management, the initial entry point is assigned a lower weight because it has a low value when compared to other nodes. Likewise, the internal escalation path to more sensitive resources is downplayed as having a low probability of ever being abused. Still, neglecting them both proves to be an explosive mix.

- **The nonmonetary costs of intrusions are hard to offset with the value contributed by healthy systems.** Loss of user confidence and business continuity, as well as the prospect of lawsuits and the risk of regulatory scrutiny, are difficult to meaningfully insure against. These effects can, at least in principle, make or break companies or even entire industries, and any superficial valuations of such outcomes are almost purely speculative.
- **Existing data is probably not representative of future risks.** Unlike the participants in a fender bender, attackers will not step forward to help- fully report break-ins and will not exhaustively document the damage caused. Unless the intrusion is painfully evident (due to the attacker's sloppiness or disruptive intent), it will often go unnoticed. Even though industry-wide, self-reported data may be available, there is simply no reli- able way of telling how complete it is or how much extra risk one's cur- rent business practice may be contributing.

* Sometime in 2006, several intruders, allegedly led by Albert Gonzalez, attacked an unsecured wireless network at a retail location and subsequently made their way through the corporate networks of the retail giant. They copied the credit card data of about 46 million customers and the Social Security numbers, home addresses, and so forth of about 450,000 more. Eleven people were charged in connection with the attack, one of whom committed suicide. † Microsoft's formally unpublished and blandly titled presentation *Threats Against and Protection of Microsoft's Internal Network* outlines a 2003 attack that began with the compromise of an engineer's home workstation that enjoyed a long-lived VPN session to the inside of the corporation. Methodical escalation attempts followed, culminating with the attacker gaining access to, and leaking data from, internal source code repositories. At least to the general public, the perpetrator remains unknown.

- **Statistical forecasting is not a robust predictor of individual outcomes.** Simply because on average people in cities are more likely to be hit by lightning than mauled by a bear does not mean you should bolt a light- ning rod to your hat and then bathe in honey. The likelihood that a compromise will be associated with a particular component is, on an individual scale, largely irrelevant: Security incidents are nearly certain, but out of thousands of exposed nontrivial resources, any service can be used as an attack vector—and no one service is likely to see a volume of events that would make statistical forecasting meaningful within the scope of a single enterprise.

Enlightenment Through Taxonomy

The two schools of thought discussed above share something in common: Both assume that it is possible to define security as a set of computable goals and that the resulting unified theory of a secure system or a model of accept- able risk would then elegantly trickle down, resulting in an optimal set of low-level actions needed to achieve perfection in application design.

Some practitioners preach the opposite approach, which owes less to philosophy and more to the natural sciences. These practitioners argue that, much like Charles Darwin of the information age, by gathering sufficient amounts of low-level, experimental data, we will be able to observe, recon- struct, and document increasingly more sophisticated laws in order to arrive some sort of a unified model of secure computing.

This latter worldview brings us projects like the Department of Home- land Security–funded Common Weakness Enumeration (CWE), the goal of which, in the organization's own words, is to develop a unified "Vulnerability Theory"; "improve the research, modeling, and classification of software flaws"; and "provide a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities." A typical, delight- fully baroque example of the resulting taxonomy may be this:

Improper Enforcement of Message or Data Structure
Failure to Sanitize Data into a Different Plane
Improper Control of Resource Identifiers
Insufficient Filtering of File and Other Resource Names for Executable Content

Today, there are about 800 names in the CWE dictionary, most of which are as discourse-enabling as the one quoted here.

A slightly different school of naturalist thought is manifested in projects such as the Common Vulnerability Scoring System (CVSS), a business-backed collaboration that aims to strictly quantify known security problems in terms of a set of basic, machine-readable parameters. A real-world example of the resulting vulnerability descriptor may be this:

AV:LN / AC:L / Au:M / C:C / I:N / A:P / E:F / RL:T / RC:UR / CDP:MH / TD:H / CR:M / IR:L / AR:M

Organizations and researchers are expected to transform this 14- dimensional vector in a carefully chosen, use-specific way in order to arrive at some sort of objective, verifiable, numerical conclusion about the signifi- cance of the underlying bug (say, "42"), precluding the need to judge the nature of security flaws in any more subjective fashion.

Yes, I am poking gentle fun at the expense of these projects, but I do not mean to belittle their effort. CWE, CVSS, and related projects serve noble goals, such as bringing a more manageable dimension to certain security pro- cesses implemented by large organizations. Still, none has yielded a grand theory of secure software, and I doubt such a framework is within sight.

Toward Practical Approaches

All signs point to security being largely a nonalgorithmic problem for now. The industry is understandably reluctant to openly embrace this notion, because it implies that there are no silver bullet solutions to preach (or better yet, commercialize); still, when pressed hard enough, eventually everybody in the security field falls back to a set of rudimentary, empirical recipes. These recipes are deeply incompatible with many business management models, but they are all that have really worked for us so far. They are as follows:

- **Learning from (preferably other people's) mistakes.** Systems should be designed to prevent known classes of bugs. In the absence of automatic (or even just elegant) solutions, this goal is best achieved by providing ongoing design guidance, ensuring that developers know what could go wrong, and giving them the tools to carry out otherwise error-prone tasks in the simplest manner possible.
- **Developing tools to detect and correct problems.** Security deficiencies typically have no obvious side effects until they're discovered by a mali- cious party: a pretty costly feedback loop. To counter this problem, we create security quality assurance (QA) tools to validate implementations and perform audits periodically to detect casual mistakes (or systemic engineering deficiencies).
- **Planning to have everything compromised.** History teaches us that major incidents will occur despite our best efforts to prevent them. It is impor- tant to implement adequate component separation, access control, data redundancy, monitoring, and response procedures so that service own- ers can react to incidents before an initially minor hiccup becomes a disaster of biblical proportions.

In all cases, a substantial dose of patience, creativity, and real technical expertise is required from all the information security staff.

Naturally, even such simple, commonsense rules—essentially basic engi- neering rigor—are often dressed up in catchphrases, sprinkled liberally with a selection of acronyms (such as *CIA*: *confidentiality*, *integrity*, *availability*), and then called "methodologies." Frequently, these methodologies are thinly veiled attempts to pass off one of the most frustrating failures of the security industry as yet another success story and, in the end, sell another cure-all

product or certification to gullible customers. But despite claims to the con- trary, such products are no substitute for street smarts and technical prow- ess—at least not today.

In any case, through the remainder of this book, I will shy away from attempts to establish or reuse any of the aforementioned grand philosophi- cal frameworks and settle for a healthy dose of anti-intellectualism instead. I will review the exposed surface of modern browsers, discuss how to use the available tools safely, which bits of the Web are commonly misunderstood, and how to control collateral damage when things go boom.

And that is, pretty much, the best take on security engineering that I can think of.

A Brief History of the Web

The Web has been plagued by a perplexing number, and a remarkable vari- ety, of security issues. Certainly, some of these problems can be attributed to one-off glitches in specific client or server implementations, but many are due to capricious, often arbitrary design decisions that govern how the essential mechanisms operate and mesh together on the browser end.

Our empire is built on shaky foundations—but why? Perhaps due to sim- ple shortsightedness: After all, back in the innocent days, who could predict the perils of contemporary networking and the economic incentives behind today's large-scale security attacks?

Unfortunately, while this explanation makes sense for truly ancient mech- anisms such as SMTP or DNS, it does not quite hold water here: The Web is relatively young and took its current shape in a setting not that different from what we see today. Instead, the key to this riddle probably lies in the tumultu- ous and unusual way in which the associated technologies have evolved.

So, pardon me another brief detour as we return to the roots. The pre- history of the Web is fairly mundane but still worth a closer look.

Tales of the Stone Age: 1945 to 1994

Computer historians frequently cite a hypothetical desk-sized device called the Memex as one of the earliest fossil records, postulated in 1945 by Vannevar Bush.[3] Memex was meant to make it possible to create, annotate, and follow cross-document links in microfilm, using a technique that vaguely resembled modern-day bookmarks and hyperlinks. Bush boldly speculated that this sim- ple capability would revolutionize the field of knowledge management and data retrieval (amusingly, a claim still occasionally ridiculed as uneducated and naïve until the early 1990s). Alas, any useful implementation of the design was out of reach at that time, so, beyond futuristic visions, nothing much happened until transistor-based computers took center stage.

The next tangible milestone, in the 1960s, was the arrival of IBM's Generalized Markup Language (GML), which allowed for the annotation of documents with machine-readable directives indicating the function of each block of text, effectively saying "this is a header," "this is a numbered list of items," and so on. Over the next 20 years or so, GML (originally used by only
a handful of IBM text editors on bulky mainframe computers) became the foundation for Standard Generalized Markup Language (SGML), a more universal and flexible language that traded an awkward colon- and period- based syntax for a familiar angle-bracketed one.

While GML was developing into SGML, computers were growing more powerful and user friendly. Several researchers began experimenting with Bush's cross-link concept, applying it to computer-based document storage and retrieval, in an effort to determine whether it would be possible to cross- reference large sets of documents based on some sort of key. Adventurous companies and universities pursued pioneering projects such as ENQUIRE, NLS, and Xanadu, but most failed to make a lasting impact. Some common complaints about the various projects revolved around their limited practical usability, excess complexity, and poor scalability.

By the end of the decade, two researchers, Tim Berners-Lee and Dan Connolly, had begun working on a new approach to the cross-domain refer- ence challenge—one that focused on simplicity. They kicked off the project by drafting HyperText Markup Language (HTML), a bare-bones descendant of SGML, designed specifically for annotating documents with hyperlinks and basic formatting. They followed their work on HTML with the develop- ment of HyperText Transfer Protocol (HTTP), an extremely basic, dedi- cated scheme for accessing HTML resources using the existing concepts of Internet Protocol (IP) addresses, domain names, and file paths. The culmi- nation of their work, sometime between 1991 and 1993, was Tim Berners- Lee's World Wide Web (Figure 1-1), a rudimentary browser that parsed HTML and allowed users to render the resulting data on the screen, and then navigate from one page to another with a mouse click.

Figure 1-1: Tim Berners-Lee's World Wide Web

To many people, the design of HTTP and HTML must have seemed a significant regression from the loftier goals of competing projects. After all, many of the earlier efforts boasted database integration, security and digital rights management, or cooperative editing and publishing; in fact, even Berners-Lee's own project, ENQUIRE, appeared more ambitious than his current work. Yet, because of its low entry requirements, immediate usability, and unconstrained scalability (which happened to coincide with the arrival of powerful and affordable computers and the expansion of the Internet), the unassuming WWW project turned out to be a sudden hit.

All right, all right, it turned out to be a "hit" by the standards of the mid- 1990s. Soon, there were no fewer than dozens of web servers running on the Internet. By 1993, HTTP traffic accounted for 0.1 percent of all bandwidth in the National Science Foundation backbone network. The same year also witnessed the arrival of Mosaic, the first reasonably popular and sophisti- cated web browser, developed at the University of Illinois. Mosaic extended the original World Wide Web code by adding features such as the ability to embed images in HTML documents and submit user data through forms, thus paving the way for the interactive, multimedia applications of today.

Mosaic made browsing prettier, helping drive consumer adoption of the Web. And through the mid-1990s, it served as the foundation for two other browsers: Mosaic Netscape (later renamed Netscape Navigator) and Spyglass Mosaic (ultimately acquired by Microsoft and renamed Internet Explorer). A handful of competing non-Mosaic engines emerged as well, including Opera and several text-based browsers (such as Lynx and w3m). The first search engines, online newspapers, and dating sites followed soon after.

The First Browser Wars: 1995 to 1999

By the mid-1990s, it was clear that the Web was here to stay and that users were willing to ditch many older technologies in favor of the new contender. Around that time, Microsoft, the desktop software behemoth that had been slow to embrace the Internet before, became uncomfortable and began to allocate substantial engineering resources to its own browser, eventually bundling it with the Windows operating system in 1996.[*] Microsoft's actions sparked a period colloquially known as the "browser wars."

The resulting arms race among browser vendors was characterized by the remarkably rapid development and deployment of new features in the compet- ing products, a trend that often defied all attempts to standardize or even prop- erly document all the newly added code. Core HTML tweaks ranged from the silly (the ability to make text blink, a Netscape invention that became the butt of jokes and a telltale sign of misguided web design) to notable ones, such as the ability to change typefaces or embed external documents in so-called frames. Vendors released their products with embedded programming languages such as JavaScript and Visual Basic, plug-ins to execute platform-independent Java

[*] Interestingly, this decision turned out to be a very controversial one. On one hand, it could be argued that in doing so, Microsoft contributed greatly to the popularization of the Internet. On the other, it undermined the position of competing browsers and could be seen as anti-competitive. In the end, the strategy led to a series of protracted legal battles over the possible abuse of monopoly by the company, such as *United States v. Microsoft*.

or Flash applets on the user's machine, and useful but tricky HTTP extensions such as cookies. Only a limited degree of superficial compatibility, sometimes hindered by patents and trademarks,[*] would be maintained.

As the Web grew larger and more diverse, a sneaky disease spread across browser engines under the guise of fault tolerance. At first, the reasoning seemed to make perfect sense: If browser A could display a poorly designed, broken page but browser B refused to (for any reason), users would inevita- bly see browser B's failure as a bug in that product and flock in droves to the seemingly more capable client, browser A. To make sure that their browsers could display almost any web page correctly, engineers developed increas- ingly complicated and undocumented heuristics designed to second-guess the intent of sloppy webmasters, often sacrificing security and occasionally even compatibility in the process. Unfortunately, each such change further reinforced bad web design practices[†] and forced the remaining vendors to catch up with the mess to stay afloat. Certainly, the absence of sufficiently detailed, up-to-date standards did not help to curb the spread of this disease. In 1994, in order to mitigate the spread of engineering anarchy and gov- ern the expansion of HTML, Tim Berners-Lee and a handful of corporate sponsors created the World Wide Web Consortium (W3C). Unfortunately for this organization, for a long while it could only watch helplessly as the for- mat was randomly extended and tweaked. Initial W3C work on HTML 2.0 and HTML 3.2 merely tried to catch up with the status quo, resulting in half- baked specs that were largely out-of-date by the time they were released to the public. The consortium also tried to work on some novel and fairly well- thought-out projects, such as Cascading Style Sheets, but had a hard time get- ting buy-in from the vendors.

Other efforts to standardize or improve already implemented mecha- nisms, most notably HTTP and JavaScript, were driven by other auspices such as the European Computer Manufacturers Association (ECMA), the Interna- tional Organization for Standardization (ISO), and the Internet Engineering Task Force (IETF). Sadly, the whole of these efforts was seldom in sync, and some discussions and design decisions were dominated by vendors or other stakeholders who did not care much about the long-term prospects of the tech- nology. The results were a number of dead standards, contradictory advice, and several frightening examples of harmful cross-interactions between other- wise neatly designed protocols—a problem that will be particularly evident when we discuss a variety of content isolation mechanisms in Chapter 9.

The Boring Period: 2000 to 2003

As the efforts to wrangle the Web floundered, Microsoft's dominance grew as a result of its operating system–bundling strategy. By the beginning of the new decade, Netscape Navigator was on the way out, and Internet Explorer

[*] For example, Microsoft did not want to deal with Sun to license a trademark for JavaScript (a language so named for promotional reasons and not because it had anything to do with Java), so it opted to name its almost-but-not-exactly-identical version "JScript." Microsoft's official documentation still refers to the software by this name. [†] Prime examples of misguided and ultimately lethal browser features are content and character set–sniffing mechanisms, both of which will be discussed in Chapter 13.

held an impressive 80 percent market share—a number roughly comparable to what Netscape had held just five years before. On both sides of the fence, security and interoperability were the two most notable casualties of the fea- ture war, but one could hope now that the fighting was over, developers could put differences aside and work together to fix the mess.

Instead, dominance bred complacency: Having achieved its goals bril- liantly, Microsoft had little incentive to invest heavily in its browser. Although through version 5, major releases of Internet Explorer (IE) arrived yearly, it took two years for version 6 to surface, then five full years for Internet Explorer 6 to be updated to Internet Explorer 7. Without Microsoft's inter- est, other vendors had very little leverage to make disruptive changes; most sites were unwilling to make improvements that would work for only a small fraction of their visitors.

On the upside, the slowdown in browser development allowed the W3C to catch up and to carefully explore some new concepts for the future of the Web. New initiatives finalized around the year 2000 included HTML 4 (a cleaned-up language that deprecated or banned many of the redundant or politically incorrect features embraced by earlier versions) and XHTML 1.1 (a strict and well-structured XML-based format that was easier to unambiguously parse, with no proprietary heuristics allowed). The consortium also made signif- icant improvements to JavaScript's Document Object Model and to Cascading Style Sheets. Regrettably, by the end of the century, the Web was too mature to casually undo some of the sins of the old, yet too young for the security issues to be pressing and evident enough for all to see. Syntax was improved, tags were deprecated, validators were written, and deck chairs were rearranged, but the browsers remained pretty much the same: bloated, quirky, and unpredictable.

But soon, something interesting happened: Microsoft gave the world a seemingly unimportant, proprietary API, confusingly named *XMLHttpRequest*. This trivial mechanism was meant to be of little significance, merely an attempt to scratch an itch in the web-based version of Microsoft Outlook. But *XMLHttpRequest* turned out to be far more, as it allowed for largely unconstrained asynchronous HTTP communications between client-side JavaScript and the server without the need for time-consuming and disrup- tive page transitions. In doing so, the API contributed to the emergence of what would later be dubbed *web 2.0*—a range of complex, unusually respon- sive, browser-based applications that enabled users to operate on complex data sets, collaborate and publish content, and so on, invading the sacred domain of "real," installable client software in the process. Understandably, this caused quite a stir.

Web 2.0 and the Second Browser Wars: 2004 and Beyond

XMLHttpRequest, in conjunction with the popularity of the Internet and the broad availability of web browsers, pushed the Web to some new, exciting frontiers—and brought us a flurry of security bugs that impacted both individual users and businesses. By about 2002, worms and browser vulnerabili- ties had emerged as a frequently revisited theme in the media. Microsoft, by virtue of its market dominance and a relatively dismissive security posture,
took much of the resulting PR heat. The company casually downplayed the problem, but the trend eventually created an atmosphere conducive to a small rebellion.

In 2004, a new contender in the browser wars emerged: Mozilla Firefox (a community-supported descendant of Netscape Navigator) took the offen- sive, specifically targeting Internet Explorer's poor security track record and standards compliance. Praised by both IT journalists and security experts, Firefox quickly secured a 20 percent market share. While the newcomer soon proved to be nearly as plagued by security bugs as its counterpart from Redmond, its open source nature and the freedom from having to cater to stub- born corporate users allowed developers to fix issues much faster.

NOTE *Why would vendors compete so feverishly? Strictly speaking, there is no money to be made by having a particular market share in the browser world. That said, pundits have long speculated that it is a matter of power: By bundling, promoting, or demoting certain online services (even as simple as the default search engine), whoever controls the browser controls much of the Internet.*

Firefox aside, Microsoft had other reasons to feel uneasy. Its flagship prod- uct, the Windows operating system, was increasingly being used as an (expend- able?) launch pad for the browser, with more and more applications (from document editors to games) moving to the Web. This could not be good.

These facts, combined with the sudden emergence of Apple's Safari browser and perhaps Opera's advances in the world of smartphones, must have had Microsoft executives scratching their heads. They had missed the early signs of the importance of the Internet in the 1990s; surely they couldn't afford to repeat the mistake. Microsoft put some steam behind Internet Explorer development again, releasing drastically improved and somewhat more secure versions 7, 8, and 9 in rapid succession.

Competitors countered with new features and claims of even better (if still superficial) standards compliance, safer browsing, and performance improve- ments. Caught off guard by the unexpected success of *XMLHttpRequest* and quick to forget other lessons from the past, vendors also decided to experi- ment boldly with new ideas, sometimes unilaterally rolling out half-baked or somewhat insecure designs like *globalStorage* in Firefox or *httponly* cookies in Internet Explorer, just to try their luck.

To further complicate the picture, frustrated by creative differences with W3C, a group of contributors created a wholly new standards body called the Web Hypertext Application Technology Working Group (WHATWG). The WHATWG has been instrumental in the development of HTML5, the first holistic and security-conscious revision of existing standards, but it is report- edly shunned by Microsoft due to patent policy disputes.

Throughout much of its history, the Web has enjoyed a unique, highly competitive, rapid, often overly political, and erratic development model with no unifying vision and no one set of security principles. This state of affairs has left a profound mark on how browsers operate today and how secure the user data handled by browsers can be.
Chances are, this situation is not going to change anytime soon.

The Evolution of a Threat

Clearly, web browsers, and their associated document formats and communi- cation protocols, evolved in an unusual manner. This evolution may explain the high number of security problems we see, but by itself it hardly proves that these problems are unique or noteworthy. To wrap up this chapter, let's take a quick look at the very special characteristics behind the most prevalent types of online security threats and explore why these threats had no particu- larly good equivalents in the years before the Web.

The User as a Security Flaw

Perhaps the most striking (and entirely nontechnical) property of web browsers is that most people who use them are overwhelmingly unskilled. Sure, nonproficient users have been an amusing, fringe problem since the dawn of computing. But the popularity of the Web, combined with its remark- ably low barrier to entry, means we are facing a new foe: Most users simply don't know enough to stay safe.

For a long time, engineers working on general-purpose software have made seemingly arbitrary assumptions about the minimal level of computer proficiency required of their users. Most of these assumptions have been with- out serious consequences; the incorrect use of a text editor, for instance, would typically have little or no impact on system security. Incompetent users simply would not be able to get their work done, a wonderfully self-correcting issue.

Web browsers do not work this way, however. Unlike certain complicated software, they can be *successfully* used by people with virtually no computer training, people who may not even know how to use a text editor. But at the same time, browsers can be operated *safely* only by people with a pretty good understanding of computer technology and its associated jargon, including topics such as Public-Key Infrastructure. Needless to say, this prerequisite is not met by most users of some of today's most successful web applications.

Browsers still look and feel as if they were designed by geeks and for geeks, complete with occasional cryptic and inconsistent error messages, complex configuration settings, and a puzzling variety of security warnings and prompts. A notable study by Berkeley and Harvard researchers in 2006 demonstrated that casual users are almost universally oblivious to signals that surely make perfect sense to a developer, such as the presence or absence of lock icons in the status bar.[4] In another study, Stanford and Microsoft researchers reached similar conclusions when they examined the impact of the modern "green URL bar" security indicator. The mechanism, designed to offer a more intuitive alternative to lock icons, actually made it easier to trick users by teaching the audience to trust a particular shade of green, no matter where this color appeared.[5]

Some experts argue that the ineptitude of the casual user is not the fault of software vendors and hence not an engineering problem at all. Others note that when creating software so easily accessible and so widely distributed, it is irresponsible to force users to make security-critical decisions that depend on technical prowess not required to operate the program in the first place.

To blame browser vendors alone is just as unfair, however: The computing industry as a whole has no robust answers in this area, and very little research is available on how to design comparably complex user interfaces (UIs) in a bulletproof way. After all, we barely get it right for ATMs.

The Cloud, or the Joys of Communal Living

Another peculiar characteristic of the Web is the dramatically understated separation between unrelated applications and the data they process.

In the traditional model followed by virtually all personal computers over the last 15 years or so, there are very clear boundaries between high- level data objects (documents), user-level code (applications), and the oper- ating system kernel that arbitrates all cross-application communications and hardware input/output (I/O) and enforces configurable security rules should an application go rogue. These boundaries are well studied and useful for building practical security schemes. A file opened in your text editor is unlikely to be able to steal your email, unless a really unfortunate conjunction of implementation flaws subverts all these layers of separation at once.

In the browser world, this separation is virtually nonexistent: Documents and code live as parts of the same intermingled blobs of HTML, isolation between completely unrelated applications is partial at best (with all sites nominally sharing a global JavaScript environment), and many types of inter- action between sites are implicitly permitted with few, if any, flexible, browser- level security arbitration frameworks.

In a sense, the model is reminiscent of CP/M, DOS, and other principally nonmultitasking operating systems with no

robust memory protection, CPU preemption, or multiuser features. The obvious difference is that few users depended on these early operating systems to simultaneously run multiple untrusted, attacker-supplied applications, so there was no particular reason for alarm.
In the end, the seemingly unlikely scenario of a text file stealing your email is, in fact, a frustratingly common pattern on the Web. Virtually all web applications must heavily compensate for unsolicited, malicious cross-domain access and take cumbersome steps to maintain at least some separation of code and the displayed data. And sooner or later, virtually all web applications fail. Content-related security issues, such as cross-site scripting or cross-site request forgery, are extremely common and have very few counterparts in dedicated, compartmentalized client architectures.

Nonconvergence of Visions

Fortunately, the browser security landscape is not entirely hopeless, and despite limited separation between web applications, several selective secu- rity mechanisms offer rudimentary protection against the most obvious attacks. But this brings us to another characteristic that makes the Web such an inter- esting subject: There is no shared, holistic security model to grasp and live by. We are not looking for a grand vision for world peace, mind you, but simply a common set of flexible paradigms that would apply to most, if not all, of the

relevant security logic. In the Unix world, for example, the *rwx* user/group per- mission model is one such strong unifying theme. But in the browser realm? In the browser realm, a mechanism called *same-origin policy* could be considered a candidate for a core security paradigm, but only until one real- izes that it governs a woefully small subset of cross-domain interactions. That detail aside, even within its scope, it has no fewer than seven distinct varieties, each of which places security boundaries between applications in a slightly different place.* Several dozen additional mechanisms, with no relation to the same-origin model, control other key aspects of browser behavior (essen- tially implementing what each author considered to be the best approach to security controls that day).
As it turns out, hundreds of small, clever hacks do not necessarily add up to a competent security opus. The unusual lack of integrity makes it very dif- ficult even to decide where a single application ends and a different one begins. Given this reality, how does one assess attack surfaces, grant or take away permissions, or accomplish just about any other security-minded task? Too often, “by keeping your fingers crossed” is the best response we can give.
Curiously, many well-intentioned attempts to improve security by defining new security controls only make the problem worse. Many of these schemes create new security boundaries that, for the sake of elegance, do not perfectly align with the hairy juxtaposition of the existing ones. When the new controls are finer grained, they are likely to be rendered ineffective by the legacy mechanisms, offering a false sense of security; when they are more coarse grained, they may eliminate some of the subtle assurances that the Web depends on right now. (Adam Barth and Collin Jackson explore the topic of destructive interference between browser security policies in their academic work.)[6]

Cross-Browser Interactions: Synergy in Failure

The overall susceptibility of an ecosystem composed of several different soft- ware products could be expected to be equal to a simple sum of the flaws contributed by each of the applications. In some cases, the resulting expo- sure may be less (diversity improves resilience), but one would not expect it to be more.
The Web is once again an exception to the rule. The security community has discovered a substantial number of issues that cannot be attributed to any particular piece of code but that emerge as a real threat when various brows- ers try to interact with each other. No particular product can be easily singled out for blame: They are all doing their thing, and the only problem is that no one has bothered to define a common etiquette for all of them to obey.
For example, one browser may assume that, in line with its own security model, it is safe to pass certain URLs to external applications or to store or read back certain types of data from disk. For each such assumption, there likely exists at least one browser that strongly disagrees, expecting other

* The primary seven varieties, as discussed throughout Part II of this book, include the security policy for JavaScript DOM access;

XMLHttpRequest API; HTTP cookies; local storage APIs; and plug-ins such as Flash, Silverlight, or Java.

parties to follow its rules instead. The exploitability of these issues is greatly aggravated by vendors' desire to get their foot in the door and try to allow web pages to switch to their browser on the fly without the user's informed consent. For example, Firefox allows pages to be opened in its browser by registering a *firefoxurl:* protocol; Microsoft installs its own .NET gateway plug- in in Firefox; Chrome does the same to Internet Explorer via a protocol named *cf:*.

NOTE *Especially in the case of such interactions, pinning the blame on any particular party is a fool's errand. In a recent case of a bug related to* firefoxurl:*, Microsoft and half of the information security community blamed Mozilla, while Mozilla and the other half of experts blamed Microsoft.*[7] *It did not matter who was right: The result was still a very real mess.*

Another set of closely related problems (practically unheard of in the days before the Web) are the incompatibilities in superficially similar security mechanisms implemented in each browser. When the security models differ, a sound web application–engineering practice in one product may be inade- quate and misguided in another. In fact, several classes of rudimentary tasks, such as serving a user-supplied plaintext file, cannot be safely implemented in certain browsers at all. This fact, however, will not be obvious to develop- ers unless they are working in one of the affected browsers—and even then, they need to hit just the right spot.

In the end, all the characteristics outlined in this section contribute to a whole new class of security vulnerabilities that a taxonomy buff might call a *failure to account for undocumented diversity*. This class is very well populated today.

The Breakdown of the Client-Server Divide

Information security researchers enjoy the world of static, clearly assigned roles, which are a familiar point of reference when mapping security inter- actions in the otherwise complicated world. For example, we talk about Alice and Bob, two wholesome, hardworking users who want to communicate, and Mallory, a sneaky attacker who is out to get them. We then have client software (essentially dumb, sometimes rogue I/O terminals that frivolously request services) and humble servers, carefully fulfilling the clients' whim. Develop- ers learn these roles and play along, building fairly comprehensible and test- able network-computing environments in the process.

The Web began as a classical example of a proper client-server architec- ture, but the functional boundaries between client and server responsibilities were quickly eroded. The culprit is JavaScript, a language that offers the HTTP servers a way to delegate application logic to the browser ("client") side and gives them two very compelling reasons to do so. First, such a shift often results in more responsive user interfaces, as servers do not need to synchro- nously participate in each tiny UI state change imaginable. Second, server- side CPU and memory requirements (and hence service-provisioning costs) can decrease drastically when individual workstations across the globe chip in to help with the bulk of the work.

The client-server diffusion process began innocently enough, but it was only a matter of time before the first security mechanisms followed to the client side too, along with all the other mundane functionality. For example, what was the point of carefully scrubbing HTML on the server side when the data was only dynamically rendered by JavaScript on the client machine?

In some applications, this trend was taken to extremes, eventually leav- ing the server as little more than a dumb storage device and moving almost all the parsing, editing, display, and configuration tasks into the browser itself. In such designs, the dependency on a server could even be fully sev- ered by using offline web extensions such as HTML5 persistent storage.

A simple shift in where the entire application magic happens is not necessarily a big deal, but not all security responsibilities can be delegated to the client as easily. For example, even in the case of a server acting as dumb storage, clients cannot be given indiscriminate access to all the data stored on the server for other users, and they cannot be trusted to enforce access controls. In the end, because it was not desirable to keep all the application

security logic on the server side, and it was impossible to migrate it fully to the client, most applications ended up occupying some arbitrary middle ground instead, with no easily discernible and logical separation of duties between the client and server components. The resulting unfamiliar designs and application behaviors simply had no useful equivalents in the elegant and wholesome world of security role-play.
The situation has resulted in more than just a design-level mess; it has led to irreducible complexity. In a traditional client-server model with well- specified APIs, one can easily evaluate a server's behavior without looking at the client, and vice versa. Moreover, within each of these components, it is possible to easily isolate smaller functional blocks and make assumptions about their intended operation. With the new model, coupled with the opaque, one-off application APIs common on the Web, these analytical tools, and the resulting ease of reasoning about the security of a system, have been brutally taken away.
The unexpected failure of standardized security modeling and testing protocols is yet another problem that earns the Web a very special—and scary—place in the universe of information security.

Global browser market share, May 2011

Vendor	Browser Name	Market Share	
Microsoft	Internet Explorer 6	10%	52%
	Internet Explorer 7	7%	
	Internet Explorer 8	31%	
	Internet Explorer 9	4%	
Mozilla	Firefox 3	12%	22%
	Firefox 4+	10%	
Google	Chrome	13%	
Apple	Safari	7%	
Opera Software	Opera	3%	

Source: Data drawn from public Net Applications reports.[1]

PART I

ANATOMY OF THE WEB

The first part of this book focuses on the principal concepts that govern the operation of web browsers, namely, the protocols, document formats, and pro- gramming languages that make it all tick. Because all the familiar, user-visible security mechanisms employed in modern browsers are profoundly intertwined with these inner workings, the bare internals deserve a fair bit of attention before we wander off deeper into the woods.

IT STARTS WITH A URL

The most recognizable hallmark of the Web is a simple text string known as the *Uniform Resource Locator* (*URL*). Each well-formed, fully qualified URL is meant to con- clusively address and uniquely identify a single resource on a remote server (and in doing so, implement a cou- ple of related, auxiliary functions). The URL syntax is the cornerstone of the address bar, the most important user interface (UI) security indicator in every browser.

In addition to true URLs used for content retrieval, several classes of *pseudo-URLs* use a similar syntax to provide convenient access to browser-level features, including the integrated scripting engine, several special document-rendering modes, and so on. Perhaps unsurprisingly, these pseudo-URL actions can have a significant impact on the security of any site that decides to link to them.

The ability to figure out how a particular URL will be interpreted by the browser, and the side effects it will have, is one of the most basic and com- mon security tasks attempted by humans and web applications alike, but it can

be a problematic one. The generic URL syntax, the work of Tim Berners-Lee, is codified primarily in RFC 3986;[1] its practical uses on the Web are outlined in RFCs 1738,[2] 2616,[3] and a couple of other, less-significant standards. These documents are remarkably detailed, resulting in a fairly complex parsing model, but they are not precise enough to lead to harmonious, compatible implementations in all client software. In addition, individual software ven- dors have chosen to deviate from the specifications for their own reasons.

Let’s have a closer look at how the humble URL works in practice.

Uniform Resource Locator Structure

Figure 2-1 shows the format of a *fully qualified absolute URL*, one that specifies all information required to access a particular resource and that does not depend in any way on where the navigation began. In contrast, a *relative URL*, such as *../file.php?text=hello+world*, omits some of this information and must be interpreted in the context of a base URL associated with the current browsing context.

scheme:// login.password@ address:port /path/to/resource ?query_string #fragment

Scheme/protocol name

Indicator of a hierarchical URL (constant)

Credentials to access the resource (optional)

Server to retrieve the data from

Port number to connect to (optional)

Hierarchical Unix path to a resource

“Query string” parameters (optional)

“Fragment identifier” (optional)

Figure 2-1: Structure of an absolute URL

The segments of the absolute URL seem intuitive, but each comes with a set of gotchas, so let’s review them now.

Scheme Name

The *scheme name* is a case-insensitive string that ends with a single colon, indicating the protocol to be used to retrieve the resource. The official registry of valid URL schemes is maintained by the *Internet Assigned Numbers Authority* (*IANA*), a body more widely known for its management of the IP address space.[4]

IANA’s current list of valid scheme names includes several dozen entries such as *http:*, *https:*, and *ftp:*; in practice, a much broader set of schemes is informally recognized by common browsers and third-party appli- cations, some which have special security consequences. (Of particular inter- est are several types of pseudo-URLs, such as *data:* or *javascript:*, as discussed later in this chapter and throughout the remainder of this book.)

“Authority”

Before they can do any further parsing, browsers and web applications need to distinguish fully qualified absolute URLs from relative ones. The presence of a valid scheme in front of the address is meant to be the key difference, as defined in RFC 1738: In a compliant absolute URL, only the alphanumerics “+”, “-”, and “.” may appear before the required “:”. In prac- tice, however, browsers deviate from this guidance a bit. All ignore leading newlines and white spaces. Internet Explorer ignores the entire nonprintable character range of ASCII codes 0x01 to 0x1F. Chrome additionally skips 0x00, the NUL character. Most implementations also ignore newlines and tabs in the middle of scheme names, and Opera accepts high-bit characters in the string. Because of these incompatibilities, applications that depend on the abil- ity to differentiate between relative and absolute URLs must conservatively reject any anomalous syntax—but as we will soon find out, even this is not enough.

Indicator of a Hierarchical URL

In order to comply with the generic syntax rules laid out in RFC 1738, every absolute, hierarchical URL is required to contain the fixed string "//" right before the authority section. If the string is missing, the format and function of the remainder of the URL is undefined for the purpose of that specifica- tion and must be treated as an opaque, scheme-specific value.

NOTE *An example of a nonhierarchical URL is the* mailto: *protocol, used to specify email addresses and possibly a subject line (*mailto:user@example.com?subject= Hello+world*). Such URLs are passed down to the default mail client without making any further attempt to parse them.*

The concept of a generic, hierarchical URL syntax is, in theory, an ele- gant one. It ought to enable applications to extract some information about the address without knowing how a particular scheme works. For example, without a preconceived notion of the *wacky-widget:* protocol, and by applying the concept of generic URL syntax alone, the browser could decide that *http://example.com/test1/* and *wacky-widget://example.com/test2/* reference the same, trusted remote host.

Regrettably, the specification has an interesting flaw: The aforementioned RFC says nothing about what the implementer should do when encountering URLs where the scheme is known to be nonhierarchical but where a "//" prefix still appears, or vice versa. In fact, a reference parser implementation provided in RFC 1630 contains an unintentional loophole that gives a counter- intuitive meaning to the latter class of URLs. In RFC 3986, published some years later, the authors sheepishly acknowledge this flaw and permit imple- mentations to try to parse such URLs for compatibility reasons. As a conse- quence, many browsers interpret the following examples in unexpected ways:

- ***http:example.com/*** In Firefox, Chrome, and Safari, this address may be treated identically to *http://example.com/* when no fully qualified base URL context exists and as a relative reference to a directory named *example.com* when a valid base URL is available.

- ***javascript://example.com/%0Aalert(1)*** This string is interpreted as a valid nonhierarchical pseudo-URL in all modern browsers, and the JavaScript *alert(1)* code will execute, showing a simple dialog window.
- ***mailto://user@example.com*** Internet Explorer accepts this URL as a valid nonhierarchical reference to an email address; the "//" part is simply skipped. Other browsers disagree.

Credentials to Access the Resource

The credentials portion of the URL is optional. This location can specify a username, and perhaps a password, that may be required to retrieve the data from the server. The method through which these credentials are exchanged is not specified as a part of the abstract URL syntax, and it is always protocol specific. For those protocols that do not support authentication, the behav- ior of a credential-bearing URL is simply undefined.

When no credentials are supplied, the browser will attempt to fetch the resource anonymously. In the case of HTTP and several other protocols, this means not sending any authentication data; for FTP, it involves logging into a guest account named *ftp* with a bogus password.

Most browsers accept almost any characters, other than general URL section delimiters, in this section with two exceptions: Safari, for unclear rea- sons, rejects a broader set of characters, including "<", ">", "{", and "}", while Firefox also rejects newlines.*

Server Address

For all fully qualified hierarchical URLs, the server address section must spec- ify a case-insensitive DNS name (such as *example.com*), a raw IPv4 address (such as *127.0.0.1*), or an IPv6 address in square brackets (such as *[0:0:0:0:0:0:0:1]*), indicating the location of a server hosting the requested resource. Firefox will also accept IPv4 addresses and hostnames in square brackets, but other implementations reject them immediately.

Although the RFC permits only canonical notations for IP addresses, stan- dard C libraries used by most applications are much more relaxed, accepting noncanonical IPv4 addresses that mix octal, decimal, and hexadecimal nota- tion or concatenate some or all of the octets into a single integer. As a result, the following options are recognized as equivalent:

- ***http://127.0.0.1/*** This is a canonical representation of an IPv4 address.
- ***http://0x7f.1/*** This is a representation of the same address that uses a hexadecimal number to represent the first octet and concatenates all the remaining octets into a single decimal value.
- ***http://017700000001/*** The same address is denoted using a 0-prefixed octal value, with all octets concatenated into a single 32-bit integer.

* This is possibly out of the concern for FTP, which transmits user credentials without any encoding; in this protocol, a newline transmitted as is would be misinterpreted by the server as the beginning of a new FTP command. Other browsers may transmit FTP credentials in noncompliant percent-encoded form or simply strip any problematic characters later on.

A similar laid-back approach can be seen with DNS names. Theoretically, DNS labels need to conform to a very narrow character set (specifically, alpha- numerics, ".", and "-", as defined in RFC 1035[5]), but many browsers will happily ask the underlying operating system resolver to look up almost anything, and the operating system will usually also not make a fuss. The exact set of charac- ters accepted in the hostname and passed to the resolver varies from client to client. Safari is most rigorous, while Internet Explorer is the most permissive. Perhaps of note, several control characters in the 0x0A–0x0D and 0xA0–0xAD ranges are ignored by most browsers in this portion of the URL.

NOTE *One fascinating behavior of the URL parsers in all of the mainstream browsers is their willingness to treat the character " " (ideographic full stop, Unicode point U+3002) identically to a period in hostnames but not anywhere else in the URL. This is report- edly because certain Chinese keyboard mappings make it much easier to type this symbol than the expected 7-bit ASCII value.*

Server Port

This server port is an optional section that describes a nonstandard network port to connect to on the previously specified server. Virtually all application- level protocols supported by browsers and third-party applications use TCP or UDP as the underlying transport method, and both TCP and UDP rely on 16-bit port numbers to separate traffic between unrelated services running on a single machine. Each scheme is associated with a default port on which servers for that protocol are customarily run (80 for HTTP, 21 for FTP, and so on), but the default can be overridden at the URL level.

NOTE *An interesting and unintended side effect of this feature is that browsers can be tricked into sending attacker-supplied data to random network services that do not speak the protocol the browser expects them to. For example, one may point a browser to* http:// mail.example.com:25/, *where 25 is a port used by the Simple Mail Transfer Protocol (SMTP) service rather than HTTP. This fact has caused a range of security problems and prompted a number of imperfect workarounds, as discussed in more detail in Part II of this book.*

Hierarchical File Path

The next portion of the URL, the hierarchical file path, is envisioned as a way to identify a specific resource to be retrieved from the server, such as */documents/2009/my_diary.txt*. The specification quite openly builds on top of the Unix directory semantics, mandating the resolution of "/../" and "/./" segments in the path and providing a directory-based method for sorting out relative references in non–fully qualified URLs.

Using the filesystem model must have seemed like a natural choice in the 1990s, when web servers acted as simple gateways to a collection of static files and the occasional executable script. But since then, many contempo- rary web application frameworks have severed any remaining ties with the filesystem, interfacing directly with database objects or registered locations in resident program code. Mapping these data structures to well-behaved URL

paths is possible but not always practiced or practiced carefully. All of this makes automated content retrieval, indexing, and security testing more complicated than it should be.

Query String

The query string is an optional section used to pass arbitrary, nonhierarchi- cal parameters to the resource earlier identified by the path. One common example is passing user-supplied terms to a server-side script that implements the search functionality, such as:

http://example.com/`search.php?query=Hello+world`

Most web developers are accustomed to a particular layout of the query string; this familiar format is generated by browsers when handling HTML- based forms and follows this syntax:

name1=value1&name2=value2...

Surprisingly, such layout is not mandated in the URL RFCs. Instead, the query string is treated as an opaque blob of data that may be interpreted by the final recipient as it sees fit, and unlike the path, it is not encumbered with specific parsing rules.

Hints of the commonly used format can be found in an informational RFC 1630,[6] in a mail-related RFC 2368,[7] and in HTML specifications dealing with forms.[8] None of this is binding, and therefore, while it may be impolite, it is not a mistake for web applications to employ arbitrary formats for what- ever data they wish to put in that part of the URL.

Fragment ID The fragment ID is an opaque value with a role similar to the query string but that provides optional instructions for the client application rather than the server. (In fact, the value is not supposed to be sent to the server at all.) Neither the format nor function of the fragment ID is clearly specified in the RFCs, but it is hinted that it may be used to address “subresources” in the retrieved document or to provide other document-specific rendering cues.

In practice, fragment identifiers have only a single sanctioned use in the browser: that of specifying the name of an anchor HTML element for in-document navigation. The logic is simple. If an anchor name is supplied in the URL and a matching HTML tag can be located, the document will be scrolled to that location for viewing; otherwise, nothing happens. Because the information is encoded in the URL, this particular view of a lengthy doc- ument could be easily shared with others or bookmarked. In this use, the meaning of a fragment ID is limited to scrolling an existing document, so there is no need to retrieve any new data from the server when only this por- tion of the URL is updated in response to user actions.

This interesting property has led to another, more recent and completely ad hoc use of this value: to store miscellaneous state information needed by client-side scripts. For example, consider a map-browsing application that puts the currently viewed map coordinates in the fragment identifier so that it will know to resume from that same location if the link is bookmarked or shared. Unlike updating the query string, changing the fragment ID on-the- fly will not trigger a time-consuming page reload, making this data-storage trick a killer feature.

Putting It All Together Again

Each of the aforementioned URL segments is delimited by certain reserved characters: slashes, colons, question marks, and so on. To make the whole approach usable, these delimiting characters should not appear anywhere in the URL for any other purpose. With this assumption in mind, imagine a sample algorithm to split absolute URLs into the aforementioned functional parts in a manner at least vaguely consistent with how browsers accomplish this task. A reasonably decent example of such an algorithm could be:

STEP 1: Extract the scheme name.

Scan for the first “:” character. The part of the URL to its left is the scheme name. Bail out if the scheme name does not conform to the expected set of characters; the URL may need to be treated as a relative one if so.

STEP 2: Consume the hierarchical URL identifier.

The string “//” should follow the scheme name. Skip it if found; bail out if not.

NOTE *In some parsing contexts, implementations will be just as happy with zero, one, or even three or more slashes instead of two, for usability reasons. In the same vein, from its inception, Internet Explorer accepted backslashes (\) in lieu of slashes in any location in the URL, presumably to assist inexperienced users.*[*] *All browsers other than Firefox eventually followed this trend and recognize URLs such as* http:\\example.com\.

STEP 3: Grab the authority section.

Scan for the next “/”, “?”, or “#”, whichever comes first, to extract the authority section from the URL. As mentioned above, most browsers will also accept “ \” as a delimiter in place of a forward slash, which may need to be accounted for. The semicolon (;) is another acceptable authority delimiter in browsers other than Internet Explorer and Safari; the rea- son for this decision is unknown.

* Unlike UNIX-derived operating systems, Microsoft Windows uses backslashes instead of slashes to delimit file paths (say, *c:\windows\system32\calc.exe*). Microsoft probably tried to compensate for the possibility that users would be confused by the need to type a different type of a slash on the Web or hoped to resolve other possible inconsistencies with *file:* URLs and similar mechanisms that would be interfacing directly with the local filesystem. Other Windows filesystem specifics (such as case insensitivity) are not replicated, however.

STEP 3A: Find the credentials, if any.

Once the authority section is extracted, locate the at symbol (@) in the substring. If found, the leading snippet constitutes login credentials, which should be further tokenized at the first occurrence of a colon (if present) to split the login and password data.

STEP 3B: Extract the destination address.

The remainder of the authority section is the destination address. Look for the first colon to separate the hostname from the port number. A special case is needed for bracket-enclosed IPv6 addresses, too.

STEP 4: Identify the path (if present).

If the authority section is followed immediately by a forward slash—or for some implementations, a backslash or semicolon, as noted earlier— scan for the next “?”, “#”, or end-of-string, whichever comes first. The text in between constitutes the path section, which should be normalized according to Unix path semantics.

STEP 5: Extract the query string (if present).

If the last successfully parsed segment is followed by a question mark, scan for the next “#” character or end-of-string, whichever comes first. The text in between is the query string.

STEP 6: Extract the fragment identifier (if present).

If the last successfully parsed segment is followed by “#”, everything from that character to the end-of-string is the fragment identifier. Either way, you’re done!

This algorithm may seem mundane, but it reveals subtle details that even seasoned programmers normally don’t think about. It also illustrates that it is extremely difficult for casual users to understand how a particular URL may be parsed. Let's start with this fairly simple case:

```
http://example.com&gibberish=1234@167772161/
```

The target of this URL—a concatenated IP address that decodes to 10.0.0.1—is not readily apparent to a nonexpert, and many users would believe they are visiting *example.com* instead.* But all right, that was an easy one! So let’s have a peek at this syntax instead:

```
http://example.com\@coredump.cx/
```

In Firefox, that URL will take the user to *coredump.cx*, because *example.com* will be interpreted as a valid value for the login field. In almost all other brows- ers, “\” will be interpreted as a path delimiter, and the user will land on *example .com* instead.

* This particular @-based trick was quickly embraced to facilitate all sorts of online fraud targeted at casual users. Attempts to mitigate its impact ranged from the heavy-handed and oddly specific (e.g., disabling URL-based authentication in Internet Explorer or crippling it with warnings in Firefox) to the fairly sensible (e.g., hostname highlighting in the address bar of several browsers).

An even more frustrating example exists for Internet Explorer. Consider this:

```
http://example.com;.coredump.cx/
```

Microsoft’s browser permits “;” in the hostname and successfully resolves this label, thanks to the appropriate configuration of the *coredump.cx* domain. Most other browsers will autocorrect the URL to *http://example.com/ ;.coredump.cx* and take the user to *example.com* instead (except for Safari, where the syntax causes an error). If this looks messy, remember that we are just getting started with how browsers work!

Reserved Characters and Percent Encoding

The URL-parsing algorithm outlined in the previous section relies on the assumption that certain reserved, syntax-delimiting characters will not appear literally in the URL in any other capacity (that is, they won't be a part of the user- name, request path, and so on). These generic, syntax-disrupting delimiters are:

: / ? # [ ] @

The RFC also names a couple of lower-tier delimiters without giving them any specific purpose, presumably to allow scheme- or application- specific features to be implemented within any of the top-level sections:

! $ & ' ( ) * + , ; =

All of the above characters are in principle off-limits, but there are legiti- mate cases where one would want to include them in the URL (for example, to accommodate arbitrary search terms entered by the user and passed to the server in the query string). Therefore, rather than ban them, the standard provides a method to encode all spurious occurrences of these values. The method, simply called *percent encoding* or *URL encoding*, substitutes characters with a percent sign (%) followed by two hexadecimal digits representing a matching ASCII value. For example, "/" will be encoded as *%2F* (uppercase is customary but not enforced). It follows that to avoid ambiguity, the naked percent sign itself must be encoded as *%25*. Any intermediaries that handle existing URLs (browsers and web applications included) are further com- pelled never to attempt to decode or encode reserved characters in relayed URLs, because the meaning of such a URL may suddenly change.

Regrettably, the immutability of reserved characters in existing URLs is at odds with the need to respond to any URLs that are technically illegal because they misuse these characters and that are encountered by the browser in the wild. This topic is not covered by the specifications at all, which forces browser vendors to improvise and causes cross-implementation inconsisten- cies. For example, should the URL *http://a@b@c/* be translated to *http:// a@b%40c/* or perhaps to *http://a%40b@c/*? Internet Explorer and Safari think the former makes more sense; other browsers side with the latter view.

The remaining characters not in the reserved set are not supposed to have any particular significance within the URL syntax itself. However, some (such as nonprintable ASCII control characters) are clearly incompatible with the idea that URLs should be human readable and transport-safe. There- fore, the RFC outlines a confusingly named subset of *unreserved* characters (consisting of alphanumerics, "-", ".", "_", and "~") and says that only this subset and the reserved characters in their intended capacity are formally allowed to appear in the URL as is.

NOTE *Curiously, these unreserved characters are only* allowed *to appear in an unescaped form; they are not* required *to do so. User agents may encode or decode them at whim, and doing so does not change the meaning of the URL at all. This property brings up yet another way to confuse users: the use of noncanonical representations of unreserved characters. Specifically, all of the following are equivalent:*

- http://example.com/
- http://%65xample.%63om/
- http://%65%78%61%6d%70%6c%65%2e%63%6f%6d/*

A number of otherwise nonreserved, printable characters are excluded from the so-called unreserved set. Because of this, strictly speaking, the RFCs require them to be unconditionally percent encoded. However, since brows- ers are not explicitly tasked with the enforcement of this rule, it is not taken very seriously. In particular, all browsers allow "^", "{", "|", and "}" to appear in URLs without escaping and will send these characters to the server as is. Internet Explorer further permits "<", ">", and "`" to go through; Internet Explorer, Firefox, and Chrome all accept "\"; Chrome and Internet Explorer will permit a double quote; and Opera and Internet Explorer both pass the nonprintable character 0x7F (DEL) as is.

Lastly, contrary to the requirements spelled out in the RFC, most brows- ers also do not encode fragment identifiers at all. This poses an unexpected challenge to client-side scripts that rely on this string and expect certain potentially unsafe characters never to appear literally. We will revisit this topic in Chapter 6.

Handling of Non-US-ASCII Text

Many languages used around the globe rely on characters outside the basic, 7-bit ASCII character set or the default 8-bit code page traditionally used by all PC-compatible systems (CP437). Heck, some languages depend on alphabets that are not based on Latin at all.

In order to accommodate the needs of an often-ignored but formidable non-English user base, various 8-bit code pages with an alternative set of high- bit characters were devised long before the emergence of the Web: ISO 8859-1,

* Similar noncanonical encodings were widely used for various types of social engineering attacks, and consequently, various countermeasures have been deployed through the years. As usual, some of these countermeasures are disruptive (for example, Firefox flat out rejects percent-encoded text in hostnames), and some are fairly good (such as the forced "canonicalization" of the address bar by decoding all the unnecessarily encoded text for display purposes).

CP850, and Windows 1252 for Western European languages; ISO 8859-2, CP852, and Windows 1250 for Eastern and Central Europe; and KOI8-R and Windows 1251 for Russia. And, because several alphabets could not be accommodated in the 256-character space, we saw the rise of complex variable- width encodings, such as Shift JIS for katakana.

The incompatibility of these character maps made it difficult to exchange documents between computers configured for different code pages. By the early 1990s, this growing problem led to the creation of *Unicode*—a sort of universal character set, too large to fit within 8 bits but meant to encompass practically all regional scripts and specialty pictographs known to man. Uni- code was followed by UTF-8, a relatively simple, variable-width representation of these characters, which was theoretically safe for all applications capable of handling traditional 8-bit formats.

Unfortunately, UTF-8 required more bytes to encode high-bit characters than did most of its competitors, and to many users, this seemed wasteful and unnecessary. Because of this criticism, it took well over a decade for UTF-8 to gain traction on the Web, and it only did so long after all the relevant protocols had solidified.

This unfortunate delay had some bearing on the handling of URLs that contain user input. Browsers needed to accommodate such use very early on, but when the developers turned to the relevant standards, they found no meaningful advice. Even years later, in 2005, the RFC 3986 had just this to say:

In local or regional contexts and with improving technology, users might benefit from being able to use a wider range of characters; such use is not defined by this specification.

Percent-encoded octets . . . may be used within a URI to represent characters outside the range of the US-ASCII coded character set if this representation is allowed by the scheme or by the protocol element in which the URI is referenced. Such a definition should specify the character encoding used to map those characters to octets prior to being percent-encoded for the URI.

Alas, despite this wishful thinking, none of the remaining standards addressed this topic. It was always possible to put raw high-bit characters in a URL, but without knowing the code page they should be interpreted in, the server would not be able to tell if that *%B1* was supposed to mean "±", "a", or some other squiggly character specific to the user's native script.

Sadly, browser vendors have not taken the initiative and come up with a consistent solution to this problem. Most browsers internally transcode URL path segments to UTF-8 (or ISO 8859-1, if sufficient), but then they generate the query string in the code page of the referring page instead. In certain cases, when URLs are entered manually or passed to certain specialized APIs, high-bit characters may be also downgraded to their 7-bit US-ASCII look- alikes, replaced with question marks, or even completely mangled due to implementation flaws.

Poorly implemented or not, the ability to pass non-English characters in query strings and paths scratched an evident itch. The traditional percent- encoding approach left just one URL segment completely out in the cold: High-bit input could not be allowed as is when specifying the name of the destination server, because at least in principle, the well-established DNS standard permitted only period-delimited alphanumerics and dashes to appear in domain names—and while nobody adhered to the rules, the set of exceptions varied from one name server to another.

An astute reader might wonder why this limitation would matter; that is, why was it important to have localized domain names in non-Latin alphabets, too? That question may be difficult to answer now. Quite simply, several folks thought a lack of these encodings would prevent businesses and individuals around the world from fully embracing and enjoying the Web—and, rightly or not, they were determined to make it happen.

This pursuit led to the formation of the Internationalized Domain Names in Applications (IDNA). First, RFC 3490,[9] which outlined a rather contrived scheme to encode arbitrary Unicode strings using alphanumerics and dashes, and then RFC 3492,[10] which described a way to apply this encoding to DNS labels using a format known as *Punycode*. Punycode looked roughly like this:

```
xn--[US-ASCII part]-[encoded Unicode data]
```

A compliant browser presented with a technically illegal URL that con- tained a literal non-US-ASCII character anywhere in the hostname was sup- posed to transform the name to Punycode before performing a DNS lookup. Consequently, when presented with Punycode in an existing URL, it should put a decoded, human-readable form of the string in the address bar.

NOTE *Combining all these incompatible encoding strategies can make for an amusing mix.*

Consider this example URL of a made-up Polish-language towel shop:

Intent: http://www.ręczniki.pl/ręcznik?model=Jaś#Złóż_zamówienie Actual URL:

http://www. .pl/r cznik?model=Ja

Label converted to Punycode

Of all the URL-based encoding approaches, IDNA soon proved to be the most problematic. In essence, the domain name in the URL shown in the browser's address bar is one of the most important security indicators on the Web, as it allows users to quickly differentiate sites they trust and have done business with from the rest of the Internet. When the hostname shown by the browser consists of 38 familiar and distinctive characters, only fairly careless victims will be tricked into thinking that their favorite *exam**p**le.com* domain and an impostor *examp**1**e.com* site are the same thing. But IDNA casually and indiscriminately extended these 38 characters to some 100,000 glyphs sup- ported by Unicode, many of which look exactly alike and are separated from each other based on functional differences alone.

Path converted to UTF-8

Query string converted to ISO 8859-2

Literal UTF-8

How bad is it? Let's consider Cyrillic, for example. This alphabet has a number of homoglyphs that look practically identical to their Latin counter- parts but that have completely different Unicode values and resolve to com- pletely different Punycode DNS names:

Latin	a	c	e	i	j	o	p	s	x	y
	U+0061	U+0063	U+0065	U+0069	U+006A	U+006F	U+0070	U+0073	U+0078	U+0079
Cyrillic	а	с	е	і	ј	о	р	ѕ	х	у
	U+0430	U+0441	U+0435	U+0456	U+0458	U+043E	U+0440	U+0455	U+0445	U+0443

When IDNA was proposed and first implemented in browsers, nobody seriously considered the consequences of this issue. Browser vendors appar- ently assumed that DNS registrars would prevent people from registering look-alike names, and registrars figured it was the browser vendors' problem to have unambiguous visuals in the address bar.

In 2002 the significance of the problem was finally recognized by all parties involved. That year, Evgeniy Gabrilovich and Alex Gontmakher pub- lished "The Homograph Attack,"[11] a paper exploring the vulnerability in great detail. They noted that any registrar-level work-arounds, even if imple- mented, would have a fatal flaw. An attacker could always purchase a whole- some top-level domain and then, on his own name server, set up a subdomain record that, with the IDNA transformation applied, would decode to a string visually identical to *example.com/* (the last character being merely a nonfunc- tional look-alike of the actual ASCII slash). The result would be:

```
http://example.com .wholesome-domain.com/
```

This only looks like a real slash.

There is nothing that a registrar can do to prevent this attack, and the ball is in the browser vendors' court. But what options do they have, exactly?

As it turns out, there aren't many. We now realize that the poorly envi- sioned IDNA standard cannot be fixed in a simple and painless way. Browser developers have responded to this risk by reverting to incomprehensible Punycode when a user's locale does not match the script seen in a particular DNS label (which causes problems when browsing foreign sites or when using imported or simply misconfigured computers); permitting IDNA only in cer- tain country-specific, top-level domains (ruling out the use of international- ized domain names in *.com* and other high-profile TLDs); and blacklisting certain "bad" characters that resemble slashes, periods, white spaces, and so forth (a fool's errand, given the number of typefaces used around the world).

These measures are drastic enough to severely hinder the adoption of internationalized domain names, probably to a point where the standard's lingering presence causes more security problems than it brings real usability benefits to non-English users.

Common URL Schemes and Their Function

Let's leave the bizarre world of URL parsing behind us and go back to the basics. Earlier in this chapter, we implied that certain schemes may have unexpected security consequences and that because of this, any web applica- tion handling user-supplied URLs must be cautious. To explain this point a bit better, it is useful to review all the URL schemes commonly supported in a typical browser environment. These can be combined into four basic groups.

Browser-Supported, Document-Fetching Protocols These schemes, handled internally by the browser, offer a way to retrieve arbitrary content using a particular transport protocol and then display it using common, browser-level rendering logic. This is the most rudimentary and the most expected function of a URL.

The list of commonly supported schemes in this category is surprisingly short: *http:* (RFC 2616), the primary transport mode used on the Web and the focus of the next chapter of this book; *https:*, an encrypted version of HTTP (RFC 2818[12]); and *ftp:*, an older file transfer protocol (RFC 959[13]). All brows- ers also support *file:* (previously also known as *local:*), a system-specific method for accessing the local filesystem or NFS and SMB shares. (This last scheme is usually not directly accessible through Internet-originating pages, though.) Two additional, obscure cases also deserve a brief mention: built-in support for the *gopher:* scheme, one of the failed predecessors of the Web (RFC 1436[14]), which is still present in Firefox, and *shttp:*, an alternative, failed take on HTTPS (RFC 2660[15]), still recognized in Internet Explorer (but today, simply aliased to HTTP).

Protocols Claimed by Third-Party Applications and Plug-ins For these schemes, matching URLs are simply dispatched to external, spe- cialized applications that implement functionality such as media playback, document viewing, or IP telephony. At this point, the involvement of the browser (mostly) ends.

Scores of external protocol handlers exist today, and it would take another thick book to cover them all. Some of the most common examples include the *acrobat:* scheme, predictably routed to Adobe Acrobat Reader; *callto:* and *sip:* schemes claimed by all sorts of instant messengers and telephony soft- ware; *daap:*, *itpc:*, and *itms:* schemes used by Apple iTunes; *mailto:*, *news:*, and *nntp:* protocols claimed by mail and Usenet clients; *mmst:*, *mmsu:*, *msbd:*, and *rtsp:* protocols for streaming media players; and so on. Browsers are some- times also included on the list. The previously mentioned *firefoxurl:* scheme launches Firefox from within another browser, while *cf:* gives access to Chrome from Internet Explorer.

For the most part, when these schemes appear in URLs, they usually have no impact on the security of the web applications that allow them to go through (although this is not guaranteed, especially in the case of plug- in–supported content). It is worth noting that third-party protocol handlers tend to be notoriously buggy and are sometimes abused to compromise the

operating system. Therefore, restricting the ability to navigate to mystery pro- tocols is a common courtesy to the user of any reasonably trustworthy website.

Nonencapsulating Pseudo-Protocols

An array of protocols is reserved to provide convenient access to the browser's scripting engine and other internal functions, without actually retrieving any remote content and perhaps without establishing an isolated document context to display the result. Many of these pseudo-protocols are highly browser-specific and are either not directly accessible from the Inter- net or are incapable of doing harm. However, there are several important exceptions to this rule.

Perhaps the best-known exception is the *javascript:* scheme (in earlier years, also available under aliases such as *livescript:* or *mocha:* in Netscape brows- ers). This scheme gives access to the JavaScript-programming engine in the context of the currently viewed website. In Internet Explorer, *vbscript:* offers similar capabilities through the proprietary Visual Basic interface.

Another important case is the *data:* protocol (RFC 2397[16]), which permits short, inline documents to be created without any extra network requests and sometimes inherits much of their operating context from the referring page. An example of a *data:* URL is:

```
data:text/plain,Why,%20hello%20there!
```

These externally accessible pseudo-URLs are of acute significance to site security. When navigated to, their payload may execute in the context of the originating domain, possibly stealing sensitive data or altering the appear- ance of the page for the affected user. We'll discuss the specific capabilities of browser scripting languages in Chapter 6, but as you might expect, they are substantial. (URL context inheritance rules, on the other hand, are the focus of Chapter 10.)

Encapsulating Pseudo-Protocols

This special class of pseudo-protocols may be used to prefix any other URL in order to force a special decoding or rendering mode for the retrieved resource. Perhaps the best-known example is the *view-source:* scheme sup- ported by Firefox and Chrome, used to display the pretty-printed source of an HTML page. This scheme is used in the following way:

```
view-source:http://www.example.com/
```

Other protocols that function similarly include *jar:*, which allows content to be extracted from ZIP files on the fly in Firefox; *wyciwyg:* and *view-cache:*, which give access to cached pages in Firefox and Chrome respectively; an oddball *feed:* scheme, which is meant to access news feeds in Safari;[17] and a host of poorly documented protocols associated with the Windows help sub- system and other components of Microsoft Windows (*hcp:*, *its:*, *mhtml:*, *mk:*, *ms-help:*, *ms-its:*, and *ms-itss:*).

The common property of many encapsulating protocols is that they allow the attacker to hide the actual URL that will be ultimately interpreted by the browser from naïve filters: view-source:javascript: (or even view-source:view-source:javascript:) followed by malicious code is a simple way to accomplish this. Some security restrictions may be present to limit such trickery, but they should not be relied upon. Another significant problem, recurring especially with Microsoft's *mhtml:*, is that using the protocol may ignore some of the content directives provided by the server on HTTP level, possibly leading to widespread misery.[18]

Closing Note on Scheme Detection

The sheer number of pseudo-protocols is the primary reason why web appli- cations need to carefully screen user-supplied URLs. The wonky and browser- specific URL-parsing patterns, coupled with the open-ended nature of the list of supported schemes, means that it is unsafe to simply blacklist known bad schemes; for example, a check for *javascript:* may be circumvented if this keyword is spliced with a tab or a newline, replaced with *vbscript:*, or prefixed with another encapsulating scheme.

Resolution of Relative URLs

Relative URLs have been mentioned on several occasions earlier in the chap- ter, and they deserve some additional attention at this point, too. The reason for their existence is that on almost every web page on the Internet, a consid- erable number of URLs will reference resources hosted on that same server, perhaps in the same directory. It would be inconvenient and wasteful to require a fully qualified URL to appear in the document every time such a reference is needed, so short, relative URLs (such as *../other_file.txt*) are used instead. The missing details are inferred from the URL of the referring document.

Because relative URLs are allowed to appear in exactly the same scenar- ios in which any absolute URL may appear, a method to distinguish between the two is necessary within the browser. Web applications also benefit from the ability to make the distinction, because most types of URL filters may want to scrutinize absolute URLs only and allow local references through as is.

The specification may make this task seem very simple: If the URL string does not begin with a valid scheme name followed by a semicolon and, pref- erably, a valid "//" sequence, it should be interpreted as a relative reference. And if no context for parsing such a relative URL exists, it should be rejected. Everything else is a safe relative link, right?

Predictably, it's not as easy as it seems. First, as outlined in previous sec- tions, the accepted set of characters in a valid scheme name, and the patterns accepted in lieu of "//", vary from one implementation to another. Perhaps more interestingly, it is a common misconception that relative links can point only to resources on the same server; quite a few other, less-obvious variants of relative URLs exist.

Let's have a quick peek at the known classes of relative URLs to better illustrate this possibility.

Scheme, but no authority present (*http:foo.txt*)

This infamous loophole is hinted at in RFC 3986 and attributed to an oversight in one of the earlier specs. While said specs descriptively clas- sified such URLs as (invalid) absolute references, they also provided a promiscuous reference-parsing algorithm keen on interpreting them incorrectly.

In the latter interpretation, these URLs would set a new protocol and path, query, or fragment ID but have the authority section copied over from the referring location. This syntax is accepted by several browsers, but inconsistently. For example, in some cases, *http:foo.txt* may be treated as a relative reference, while *https:example.com* may be parsed as an absolute one!

No scheme, but authority present (*//example.com*)

This is another notoriously confusing but at least well-documented quirk. While */example.com* is areference to a local resource on the current server, the standard compels browsers to treat *//example.com* as a very different case: a reference to a different authority over the current protocol. In this scenario, the scheme will be copied over from the referring location, and all other URL details will be derived from the relative URL.

No scheme, no authority, but path present (*../notes.txt*)

This is the most common variant of a relative link. Protocol and author- ity information is copied over from the referring URL. If the relative URL does not start with a slash, the path will also be copied over up to the rightmost "/". For example, if the base URL is *http://www.example .com/files/*, the path is the same, but in *http://www.example.com/files/index .html*, the filename is truncated. The new path is then appended, and standard path normalization follows on the concatenated value. The query string and fragment ID are derived only from the relative URL.

No scheme, no authority, no path, but query string present (*?search=bunnies*)

In this scenario, protocol, authority, and path information are copied verbatim from the referring URL. The query string and fragment ID are derived from the relative URL.

Only fragment ID present (#*bunnies*)

All information except for the fragment ID is copied verbatim from the referring URL; only the fragment ID is substituted. Following this type of relative URL does not cause the page to be reloaded under normal cir- cumstances, as noted earlier.

Because of the risk of potential misunderstandings between application- level URL filters and the browser when handling these types of relative refer- ences, it is a good design practice never to output user-supplied relative URLs verbatim. Where feasible, they should be explicitly rewritten to absolute ref- erences, and all security checks should be carried out against the resulting fully qualified address instead.

Security Engineering Cheat Sheet

When Constructing Brand-New URLs Based on User Input

☑ **If you allow user-supplied data in path, query, or fragment ID:** If one of the section
delimiters manages to get through without proper escaping, the URL may have a differ- ent effect from what you intended (for example, linking one of the user-visible HTML buttons to the wrong server-side action). It is okay to err on the side of caution: When inserting an attacker-controlled field value, you can simply percent-escape everything but alphanumerics. ☑ **If you allow user-supplied scheme name or authority section:** This is a major code injec-
tion and phishing risk! Apply the relevant input-validation rules outlined below.

When Designing URL Input Filters

☑ **Relative URLs:** Disallow or explicitly rewrite them to absolute references to avoid trouble.
Anything else is very likely unsafe. ☑ **Scheme name:** Permit only known prefixes, such as *http://*, *https://*, or *ftp://*. Do not use
blacklisting instead; it is extremely unsafe. ☑ **Authority section:** Hostname should contain only alphanumerics, "-", and "." and can only
be followed by "/", "?", "#", or end-of-string. Allowing anything else will backfire. If you need to examine the hostname, make sure to make a proper right-hand substring match. In rare cases, you might need to account for IDNA, IPv6 bracket notation, port num- bers, or HTTP credentials in the URL. If so, you must fully parse the URL, validate all sec- tions and reject anomalous values, and reserialize them into a nonambiguous, canonical, well-escaped representation.

When Decoding Parameters Received Through URLs

☑ Do not assume that any particular character will be escaped just because the standard says
so or because your browser does it. Before echoing back any URL-derived values or put- ting them inside database queries, new URLs, and so on, scrub them carefully for danger- ous characters.

HYPERTEXT TRANSFER PROTOCOL

The next essential concept we need to discuss is the Hypertext Transfer Protocol (HTTP): the core trans- fer mechanism of the Web and the preferred method for exchanging URL-referenced documents between servers and clients. Despite having hypertext in its name, HTTP and the actual hypertext content (the HTML language) often exist independent of each other. That said, they are intertwined in sometimes surprising ways.

The history of HTTP offers interesting insight into its authors' ambitions and the growing relevance of the Internet. Tim Berners-Lee's earliest 1991 draft of the protocol (HTTP/0.9[1]) was barely one and a half pages long, and it failed to account for even the most intuitive future needs, such as extensi- bility needed to transmit non-HTML data.

Five years and several iterations of the specification later, the first official HTTP/1.0 standard (RFC 1945[2]) tried to rectify many of these short- comings in about 50 densely packed pages of text. Fast-forward to 1999, and in HTTP/1.1 (RFC 2616[3]), the seven credited authors attempted to antici- pate almost every possible use of the protocol, creating an opus over 150 pages long. That's not all: As of this writing, the current work on HTTPbis,[4] essentially a replacement for the HTTP/1.1 specification, comes to 360 pages or so. While much of the gradually accumulated content is irrelevant to the modern Web, this progression makes it clear that the desire to tack on new features far outweighs the desire to prune failed ones.

Today, all clients and servers support a not-entirely-accurate superset of HTTP/1.0, and most can speak a reasonably complete dialect of HTTP/1.1, with a couple of extensions bolted on. Despite the fact that there is no practi- cal need to do so, several web servers, and all common browsers, also main- tain backward compatibility with HTTP/0.9.

Basic Syntax of HTTP Traffic

At a glance, HTTP is a fairly simple, text-based protocol built on top of TCP/IP.* Every HTTP session is initiated by establishing a TCP connection to the server, typically to port 80, and then issuing a request that outlines the requested URL. In response, the server returns the requested file and, in the most rudimentary use case, tears down the TCP connection immediately thereafter.

The original HTTP/0.9 protocol provided no room for any additional metadata to be exchanged between the participating parties. The client request always consisted of a single line, starting with GET, followed by the URL path and query string, and ending with a single CRLF newline (ASCII characters 0x0D 0x0A; servers were also advised to accept a lone LF). A sample HTTP/0.9 request might have looked like this:

```
GET /fuzzy_bunnies.txt
```

In response to this message, the server would have immediately returned the appropriate HTML payload. (The specification required servers to wrap lines of the returned document at 80 characters, but this advice wasn't really followed.)

The HTTP/0.9 approach has a number of substantial deficiencies. For example, it offers no way for browsers to communicate users' language pref- erences, supply a list of supported document types, and so on. It also gives servers no way to tell a client that the requested file could not be found, that it has moved to a different location, or that the returned file is not an HTML

* *Transmission Control Protocol (TCP)* is one of the core communications protocols of the Internet, providing the transport layer to any application protocols built on top of it. TCP offers reason- ably reliable, peer-acknowledged, ordered, session-based connectivity between networked hosts. In most cases, the protocol is also fairly resilient against blind packet spoofing attacks attempted by other, nonlocal hosts on the Internet.

document to begin with. Finally, the scheme is not kind to server admin- istrators: When the transmitted URL information is limited to only the path and query strings, it is impossible for a server to host multiple websites, distinguished by their hostnames, under one IP address—and unlike DNS records, IP addresses don't come cheap.

In order to fix these shortcomings (and to make room for future tweaks), HTTP/1.0 and HTTP/1.1 standards embrace a slightly different conversation format: The first line of a request is modified to include proto- col version information, and it is followed by zero or more *name: value* pairs (also known as *headers*), each occupying a separate line. Common request headers included in such requests are *User-Agent* (browser version informa- tion), *Host* (URL hostname), *Accept* (supported MIME document types*), *Accept-Language* (supported language codes), and *Referer* (a misspelled field indicating the originating page for the request, if known).

These headers are terminated with a single empty line, which may be followed by any payload the client wishes to pass to the server (the length of which must be explicitly specified with an additional *Content-Length* header). The contents of the payload are opaque from the perspective of the protocol itself; in HTML, this location is commonly used for submitting form data in one of several possible formats, though this is in no way a requirement.

Overall, a simple HTTP/1.1 request may look like this:

```
POST /fuzzy_bunnies/bunny_dispenser.php HTTP/1.1 Host: www.fuzzybunnies.com User-Agent: Bunny-Browser/1.7
Content-Type: text/plain Content-Length: 17 Referer: http://www.fuzzybunnies.com/main.html
I REQUEST A BUNNY
```

The server is expected to respond to this query by opening with a line that specifies the supported protocol version, a numerical status code (used to indicate error conditions and other special circumstances), and an optional, human-readable status message. A set of self-explanatory headers comes next, ending with an empty line. The response continues with the contents of the requested resource:

```
HTTP/1.1 200 OK Server: Bunny-Server/0.9.2 Content-Type: text/plain Connection: close
BUNNY WISH HAS BEEN GRANTED
```

* MIME type (aka *Internet media type*) is a simple, two-component value identifying the class and format of any given computer file. The concept originated in RFC 2045 and RFC 2046, where it served as a way to describe email attachments. The registry of official values (such as *text/plain* or *audio/mpeg*) is currently maintained by IANA, but ad hoc types are fairly common.

RFC 2616 also permits the response to be compressed in transit using one of three supported methods (*gzip*, *compress*, *deflate*), unless the client explicitly opts out by providing a suitable *Accept-Encoding* header.

The Consequences of Supporting HTTP/0.9

Despite the improvements made in HTTP/1.0 and HTTP/1.1, the unwelcome legacy of the "dumb" HTTP/0.9 protocol lives on, even if it is normally hid- den from view. The specification for HTTP/1.0 is partly to blame for this, because it requested that all future HTTP clients and servers support the original, half-baked draft. Specifically, section 3.1 says:

HTTP/1.0 clients must . . . understand any valid response in the format of HTTP/0.9 or HTTP/1.0.

In later years, RFC 2616 attempted to backtrack on this requirement (section 19.6: "It is beyond the scope of a protocol specification to mandate compliance with previous versions."), but acting on the earlier advice, all modern browsers continue to support the legacy protocol as well.

To understand why this pattern is dangerous, recall that HTTP/0.9 serv- ers reply with nothing but the requested file. There is no indication that the responding party actually understands HTTP and wishes to serve an HTML document. With this in mind, let's analyze what happens if the browser sends an HTTP/1.1 request to an unsuspecting SMTP service running on port 25 of *example.com*:

```
GET /<html><body><h1>Hi! HTTP/1.1 Host: example.com:25 ...
```

Because the SMTP server doesn't understand what is going on, it's likely to respond this way:

```
220 example.com ESMTP 500 5.5.1 Invalid command: "GET /<html><body><h1>Hi! HTTP/1.1" 500 5.1.1 Invalid command: "Host: example.com:25" ... 421 4.4.1 Timeout
```

All browsers willing to follow the RFC are compelled to accept these messages as the body of a valid HTTP/0.9 response and assume that the returned document is, indeed, HTML. These browsers will interpret the quoted attacker-controlled snippet appearing in one of the error messages as if it comes from the owners of a legitimate website at *example.com*. This profoundly interferes with the browser security model discussed in Part II of this book and, therefore, is pretty bad.

Newline Handling Quirks

Setting aside the radical changes between HTTP/0.9 and HTTP/1.0, several other core syntax tweaks were made later in the game. Perhaps most notably, contrary to the letter of earlier iterations, HTTP/1.1 asks clients not only to honor newlines in the CRLF and LF format but also to recognize a lone CR character. Although this recommendation is disregarded by the two most popular web servers (IIS and Apache), it is followed on the client side by all browsers except Firefox.

The resulting inconsistency makes it easier for application developers to forget that not only LF but also CR characters must be stripped from any attacker-controlled values that appear anywhere in HTTP headers. To illus- trate the problem, consider the following server response, where a user- supplied and insufficiently sanitized value appears in one of the headers, as highlighted in bold:

```
HTTP/1.1 200 OK[CR][LF] Set-Cookie: last_search_term=[CR][CR]<html><body><h1>Hi![CR][LF] [CR][LF] Action completed.
```

To Internet Explorer, this response may appear as:

```
HTTP/1.1 200 OK Set-Cookie: last_search_term=
<html><body><h1>Hi!
```

```
Action completed.
```

In fact, the class of vulnerabilities related to HTTP header newline smuggling—be it due to this inconsistency or just due to a failure to filter any type of a newline—is common enough to have its own name: *header injection* or *response splitting*.

Another little-known and potentially security-relevant tweak is support for multiline headers, a change introduced in HTTP/1.1. According to the standard, any header line that begins with a whitespace is treated as a contin- uation of the previous one. For example:

```
X-Random-Comment: This is a very long string,
so why not wrap it neatly?
```

Multiline headers are recognized in client-issued requests by IIS and Apache, but they are not supported by Internet Explorer, Safari, or Opera. Therefore, any implementation that relies on or simply permits this syntax in any attacker-influenced setting may be in trouble. Thankfully, this is rare.

Proxy Requests

Proxies are used by many organizations and Internet service providers to intercept, inspect, and forward HTTP requests on behalf of their users. This may be done to improve performance (by allowing certain server responses to be cached on a nearby system), to enforce network usage policies (for example, to prevent access to porn), or to offer monitored and authenti- cated access to otherwise separated network environments.

Conventional HTTP proxies depend on explicit browser support: The application needs to be configured to make a modified request to the proxy system, instead of attempting to talk to the intended destination. To request an HTTP resource through such a proxy, the browser will normally send a request like this:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1 User-Agent: Bunny-Browser/1.7 Host: www.fuzzybunnies.com ...
```

The key difference between the above example and the usual syntax is the presence of a fully qualified URL in the first line of the request (*http:// www.fuzzybunnies.com/*), instructing the proxy where to connect to on behalf of the user. This information is somewhat redundant, given that the *Host* header already specifies the hostname; the only reason for this overlap is that the mechanisms evolved independent of each other. To avoid being fooled by co-conspiring clients and servers, proxies should either correct any mis- matching *Host* headers to match the request URL or associate cached con- tent with a particular URL-*Host* pair and not just one of these values.

Many HTTP proxies also allow browsers to request non-HTTP resources, such as FTP files or directories. In these cases, the proxy will wrap the response in HTTP, and perhaps convert it to HTML if appropriate, before returning it to the user.* That said, if the proxy does not understand the requested proto- col, or if it is simply inappropriate for it to peek into the exchanged data (for example, inside encrypted sessions), a different approach must be used. A special type of a request, CONNECT, is reserved for this purpose but is not further explained in the HTTP/1.1 RFC. The relevant request syntax is instead outlined in a separate, draft-only specification from 1998.[5] It looks like this:

```
CONNECT www.fuzzybunnies.com:1234 HTTP/1.1 User-Agent: Bunny-Browser/1.7 ...
```

* In this case, some HTTP headers supplied by the client may be used internally by the proxy, but they will not be transmitted to the non-HTTP endpoint, which creates some interesting, if non-security-relevant, protocol ambiguities.

If the proxy is willing and able to connect to the requested destination, it acknowledges this request with a specific HTTP response code, and the role of this protocol ends. At that point, the browser will begin sending and receiv- ing raw binary data within the established TCP stream; the proxy, in turn, is expected to forward the traffic between the two endpoints indiscriminately.

NOTE *Hilariously, due to a subtle omission in the draft spec, many browsers have incorrectly processed the nonencrypted, proxy-originating error responses returned during an attempt to establish an encrypted connection. The affected implementations interpreted such plaintext responses as though they originated from the destination server over a secure channel. This glitch effectively eliminated all assurances associated with the use of encrypted communications on the Web. It took over a decade to spot and correct the flaw.*[6]

Several other classes of lower-level proxies do not use HTTP to com- municate directly with the browser but nevertheless inspect the exchanged HTTP messages to cache content or enforce certain rules. The canonical example of this is a transparent proxy that silently intercepts traffic at the TCP/IP level. The approach taken by transparent proxies is unusually dan- gerous: Any such proxy can look at the destination IP and the *Host* header sent in the intercepted connection, but it has no way of immediately telling if that destination IP is genuinely associated with the specified server name. Unless an additional lookup and correlation is performed, co-conspiring cli- ents and servers can have a field day with this behavior. Without these addi- tional checks, the attacker simply needs to connect to his or her home server and send a misleading *Host: www.google.com* header to have the response cached for all other users as though genuinely coming from *www.google.com*.

Resolution of Duplicate or Conflicting Headers

Despite being relatively verbose, RFC 2616 does a poor job of explaining how a compliant parser should resolve potential ambiguities and conflicts in the request or response data. Section 19.2 of this RFC ("Tolerant Applications") recommends relaxed and error-tolerant parsing of certain fields in "unam- biguous" cases, but the meaning of the term itself is, shall we say, not particu- larly unambiguous.

For example, because of a lack of specification-level advice, roughly half of all browsers will favor the first occurrence of a particular HTTP header, and the rest will favor the last one, ensuring that almost every header injec- tion vulnerability, no matter how constrained, is exploitable for at least some percentage of targeted users. On the server side, the situation is similarly ran- dom: Apache will honor the first *Host* header seen, while IIS will completely reject a request with multiple instances of this field.

On a related note, the relevant RFCs contain no explicit prohibition on mixing potentially conflicting HTTP/1.0 and HTTP/1.1 headers and no requirement for HTTP/1.0 servers or clients to ignore all HTTP/1.1 syntax. Because of this design, it is difficult to predict the outcome of indirect con- flicts between HTTP/1.0 and HTTP/1.1 directives that are responsible for the same thing, such as *Expires* and *Cache-Control*.

Finally, in some rare cases, header conflict resolution is outlined in the spec very clearly, but the purpose of permitting such conflicts to arise in the first place is much harder to understand. For example, HTTP/1.1 clients are required to send the *Host* header on all requests, but servers (not just prox- ies!) are also required to recognize absolute URLs in the first line of the request, as opposed to the traditional path- and query-only method. This rule permits a curiosity such as this:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1 Host: www.bunnyoutlet.com
```

In this case, section 5.2 of RFC 2616 instructs clients to disregard the nonfunctional (but still mandatory!) *Host* header, and many implementa- tions follow this advice. The problem is that underlying applications are likely to be unaware of this quirk and may instead make somewhat important deci- sions based on the inspected header value.

NOTE *When complaining about the omissions in the HTTP RFCs, it is important to recognize that the alternatives can be just as problematic. In several scenarios outlined in that RFC, the desire to explicitly mandate the handling of certain corner cases led to patently absurd outcomes. One such example is the advice on parsing dates in certain HTTP headers, at the request of section 3.3 in RFC 1945. The resulting implementation (the* prtime.c *file in the Firefox codebase*[7]*) consists of close to 2,000 lines of extremely con- fusing and unreadable C code just to decipher the specified date, time, and time zone in a sufficiently fault-tolerant way (for uses such as deciding cache content expiration).*

Semicolon-Delimited Header Values

Several HTTP headers, such as *Cache-Control* or *Content-Disposition*, use a semicolon-delimited syntax to cram several separate *name=value* pairs into a single line. The reason for allowing this nested notation is unclear, but it is probably driven by the belief that it will be a more efficient or a more intuitive approach that using several separate headers that would always have to go hand in hand.

Some use cases outlined in RFC 2616 permit *quoted-string* as the right- hand parameter in such pairs. *Quoted-string* is a syntax in which a sequence of arbitrary printable characters is surrounded by double quotes, which act as delimiters. Naturally, the quote mark itself cannot appear inside the string, but—importantly—a semicolon or a whitespace may, permitting many other- wise problematic values to be sent as is.

Unfortunately for developers, Internet Explorer does not cope with the *quoted-string* syntax particularly well, effectively rendering this encoding scheme useless. The browser will parse the following line (which is meant to indicate that the response is a downloadable file rather than an inline docu- ment) in an unexpected way:

```
Content-Disposition: attachment; filename="evil_file.exe;.txt"
```

In Microsoft's implementation, the filename will be truncated at the semicolon character and will appear to be *evil_file.exe*. This behavior creates a potential hazard to any application that relies on examining or appending a "safe" filename extension to an attacker-controlled filename and otherwise correctly checks for the quote character and newlines in this string.

NOTE *An additional* quoted-pair *mechanism is provided to allow quotes (and any other char- acters) to be used safely in the string when prefixed by a backslash. This mechanism appears to be specified incorrectly, however, and not supported by any major browser except for Opera. For* quoted-pair *to work properly, stray "\" characters would need to be banned from the* quoted-string*, which isn't the case in RFC 2616.* Quoted-pair *also permits any* CHAR*-type token to be quoted, including newlines, which is incom- patible with other HTTP-parsing rules.*

It is also worth noting that when duplicate semicolon-delimited fields are found in a single HTTP header, their order of precedence is not defined by the RFC. In the case of *filename=* in *Content-Disposition*, all mainstream browsers use the first occurrence. But there is little consistency elsewhere. For example, when extracting the *URL=* value from the *Refresh* header (used to force reload- ing the page after a specified amount of time), Internet Explorer 6 will fall back to the last instance, yet all other browsers will prefer the first one. And when handling *Content-Type*, Internet Explorer, Safari, and Opera will use the first *charset=* value, while Firefox and Chrome will rely on the last.

NOTE *Food for thought: A fascinating but largely non-security-related survey of dozens of inconsistencies associated with the handling of just a single HTTP header—* Content-Disposition*—can be found on a page maintained by Julian Reschke:* http://greenbytes.de/tech/tc2231/.

Header Character Set and Encoding Schemes

Like the documents that laid the groundwork for URL handling, all subse- quent HTTP specs have largely avoided the topic of dealing with non-US- ASCII characters inside header values. There are several plausible scenarios where non-English text may legitimately appear in this context (for example, the filename in *Content-Disposition*), but when it comes to this, the expected browser behavior is essentially undefined.

Originally, RFC 1945 permitted the TEXT token (a primitive broadly used to define the syntax of other fields) to contain 8-bit characters, provid- ing the following definition:

```
OCTET = <any 8-bit sequence of data> CTL = <any US-ASCII control character (octets 0 - 31) and DEL (127)> TEXT = <any
                                                                                                  OCTET except CTLs,
but including LWS>
```

The RFC followed up with cryptic advice: When non-US-ASCII charac- ters are encountered in a TEXT field, clients and servers *may* interpret them as ISO-8859-1, the standard Western European code page, but they don't have to. Later, RFC 2616 copied and pasted the same specification of TEXT tokens but added a note that non-ISO-8859-1 strings must be encoded using a format outlined in RFC 2047,[8] originally created for email communications. Fair enough; in this simple scheme, the encoded string opens with a "=?" pre- fix, followed by a character-set name, a "?q?" or "?b?" encoding-type indicator (*quoted-printable*[*] or *base64,*[†] respectively), and lastly the encoded string itself. The sequence ends with a "?=" terminator. An example of this may be:

```
Content-Disposition: attachment; filename="=?utf-8?q?Hi=21.txt?="
```

NOTE *The RFC should also have stated that any spurious "=?...?=" patterns must never be allowed as is in the relevant headers, in order to avoid unintended decoding of values that were not really encoded*

to begin with.
Sadly, the support for this RFC 2047 encoding is spotty. It is recognized in some headers by Firefox and Chrome, but other browsers are less cooper- ative. Internet Explorer chooses to recognize URL-style percent encoding in the *Content-Disposition* field instead (a habit also picked up by Chrome) and defaults to UTF-8 in this case. Firefox and Opera, on the other hand, prefer supporting a peculiar percent-encoded syntax proposed in RFC 2231,[9] a striking deviation from how HTTP syntax is supposed to look:

```
Content-Disposition: attachment; filename*=utf-8'en-us'Hi%21.txt
```

Astute readers may notice that there is no single encoding scheme sup- ported by all browsers at once. This situation prompts some web application developers to resort to using raw high-bit values in the HTTP headers, typi- cally interpreted as UTF-8, but doing so is somewhat unsafe. In Firefox, for example, a long-standing glitch causes UTF-8 text to be mangled when put

* *Quoted-printable* is a simple encoding scheme that replaces any nonprintable or otherwise illegal characters with the equal sign (=) followed by a 2-digit hexadecimal representation of the 8-bit character value to be encoded. Any stray equal signs in the input text must be replaced with "=3D" as well. † *Base64* is a non-human-readable encoding that encodes arbitrary 8-bit input using a 6-bit alpha- bet of case-sensitive alphanumerics, "+", and "/". Every 3 bytes of input map to 4 bytes of output. If the input does not end at a 3-byte boundary, this is indicated by appending one or two equal signs at the end of the output string.

in the *Cookie* header, permitting attacker-injected cookie delimiters to mate- rialize in unexpected places.[10] In other words, there are no easy and robust solutions to this mess.
When discussing character encodings, the problem of handling of the NUL character (0x00) probably deserves a mention. This character, used as a string terminator in many programming languages, is technically prohibited from appearing in HTTP headers (except for the aforementioned, dysfunc- tional *quoted-pair* syntax), but as you may recall, parsers are encouraged to be tolerant. When this character is allowed to go through, it is likely to have unexpected side effects. For example, *Content-Disposition* headers are trun- cated at NUL by Internet Explorer, Firefox, and Chrome but not by Opera or Safari.

Referer Header Behavior

As mentioned earlier in this chapter, HTTP requests may include a *Referer* header. This header contains the URL of a document that triggered the cur- rent navigation in some way. It is meant to help with certain troubleshooting tasks and to promote the growth of the Web by emphasizing cross-references between related web pages.
Unfortunately, the header may also reveal some information about user browsing habits to certain unfriendly parties, and it may leak sensitive infor- mation that is encoded in the URL query parameters on the referring page. Due to these concerns, and the subsequent poor advice on how to mitigate them, the header is often misused for security or policy enforcement pur- poses, but it is not up to the task. The main problem is that there is no way to differentiate between a client that is not providing the header because of user privacy preferences, one that is not providing it because of the type of navigation taking place, and one that is deliberately tricked into hiding this information by a malicious referring site.
Normally, this header is included in most HTTP requests (and preserved across HTTP-level redirects), except in the following scenarios:

- After organically entering a new URL into the address bar or opening a bookmarked page.
- When the navigation originates from a pseudo-URL document, such as *data:* or *javascript:*.
- When the request is a result of redirection controlled by the *Refresh* header (but not a *Location*-based one).
- Whenever the referring site is encrypted but the requested page isn't. According to RFC 2616 section 15.1.2, this is done for privacy reasons, but it does not make a lot of sense. The *Referer* string is still disclosed to third parties when one navigates from one encrypted domain to an unrelated encrypted one, and rest assured, the use of encryption is not synonymous with trustworthiness.
- If the user decides to block or spoof the header by tweaking browser set- tings or installing a privacy-oriented

plug-in.

As should be apparent, four out of five of these conditions can be pur- posefully induced by any rogue site.

HTTP Request Types

The original HTTP/0.9 draft provided a single method (or "verb") for requesting a document: GET. The subsequent proposals experimented with an increasingly bizarre set of methods to permit interactions other than retrieving a document or running a script, including such curiosities as SHOWMETHOD, CHECKOUT, or—why not—SPACEJUMP.[11]

Most of these thought experiments have been abandoned in HTTP/1.1, which settles on a more manageable set of eight methods. Only the first two request types—GET and POST—are of any significance to most of the mod- ern Web.

GET The GET method is meant to signify information retrieval. In practice, it is used for almost all client-server interactions in the course of a normal browsing session. Regular GET requests carry no browser-supplied payloads, although they are not strictly prohibited from doing so.

The expectation is that GET requests should not have, to quote the RFC, "significance of taking an action other than retrieval" (that is, they should make no persistent changes to the state of the application). This requirement is increasingly meaningless in modern web applications, where the applica- tion state is often not even managed entirely on the server side; consequently, the advice is widely ignored by application developers.*

NOTE *In HTTP/1.1, clients may ask the server for any set of possibly noncontiguous or over-lapping fragments of the target document by specifying the* Range *header on GET (and, less commonly, on some other types of requests). The server is not obliged to comply, but where the mechanism is available, browsers may use it to resume aborted downloads.*

POST The POST method is meant for submitting information (chiefly HTML forms) to the server for processing. Because POST actions may have persis- tent side effects, many browsers ask the user to confirm before reloading any content retrieved with POST, but for the most part, GET and POST are used in a quasi-interchangeable manner. POST requests are commonly accompanied by a payload, the length of which is indicated by the *Content-Length* header. In the case of plain HTML, the payload may consist of URL-encoded or MIME-encoded form data (a for- mat detailed in Chapter 4), although again, the syntax is not constrained at the HTTP level in any special way.

* There is an anecdotal (and perhaps even true) tale of an unfortunate webmaster by the name of John Breckman. According to the story, John's website has been accidentally deleted by a search engine–indexing robot. The robot simply unwittingly discovered an unauthenticated, GET-based administrative interface that John had built for his site . . . and happily followed every "delete" link it could find.

HEAD

HEAD is a rarely used request type that is essentially identical to GET but that returns only the HTTP headers, and not the actual payload, for the requested content. Browsers generally do not issue HEAD requests on their own, but the method is sometimes employed by search engine bots and other automated tools, for example, to probe for the existence of a file or to check its modification time.

OPTIONS

OPTIONS is a metarequest that returns the set of supported methods for a particular URL (or "*", meaning the server in general) in a response header. The OPTIONS method is almost never used in practice, except for server fingerprinting; because of its limited value, the returned information may not be very accurate.

NOTE *For the sake of completeness, we need to note that OPTIONS requests are also a corner-stone of a proposed cross-domain request authorization scheme, and as such, they may gain some prominence soon. We will revisit this scheme, and explore many other upcom- ing browser security features, in Chapter 16.*

PUT

A PUT request is meant to allow files to be uploaded to the server at the specified target URL. Because browsers do not support PUT, intentional file- upload capabilities are almost always implemented through POST to a server- side script, rather than with this theoretically more elegant approach.
That said, some nonweb HTTP clients and servers may use PUT for their own purposes. Just as interestingly, some web servers may be misconfigured to process PUT requests indiscriminately, creating an obvious security risk.

DELETE

DELETE is a self-explanatory method that complements PUT (and that is equally uncommon in practice).

TRACE

TRACE is a form of "ping" request that returns information about all the proxy hops involved in processing a request and echoes the original request as well. TRACE requests are not issued by web browsers and are seldom used for legitimate purposes. TRACE's primary use is for security testing, where it may reveal interesting details about the internal architecture of HTTP serv- ers in a remote network. Precisely for this reason, the method is often dis- abled by server administrators.

CONNECT

The CONNECT method is reserved for establishing non-HTTP connections through HTTP proxies. It is not meant to be issued directly to servers. If the support for CONNECT request is enabled accidentally on a particular server, it may pose a security risk by offering an attacker a way to tunnel TCP traffic into an otherwise protected network.

Other HTTP Methods

A number of other request methods may be employed by other nonbrowser applications or browser extensions; the most popular set of HTTP extensions may be WebDAV, an authoring and version-control protocol described in RFC 4918.[12]
Further, the *XMLHttpRequest* API nominally allows client-side JavaScript to make requests with almost arbitrary methods to the originating server— although this last functionality is heavily restricted in certain browsers (we will look into this in Chapter 9).

Server Response Codes

Section 10 of RFC 2616 lists nearly 50 status codes that a server may choose from when constructing a response. About 15 of these are used in real life, and the rest are used to indicate increasingly bizarre or unlikely states, such as "402 Payment Required" or "415 Unsupported Media Type." Most of the RFC-listed states do not map cleanly to the behavior of modern web applica- tions; the only reason for their existence is that somebody hoped they even- tually would.
A few codes are worth memorizing because they are common or carry special meaning, as discussed below.

200–299: Success

This range of status codes is used to indicate a successful completion of a request:
200 OK This is a normal response to a successful GET or POST. The browser will display the subsequently returned payload to the user or will process it in some other context-specific way. **204 No Content** This code is sometimes used to indicate a successful request to which no verbose response is expected. A 204 response aborts navigation to the URL that triggered it and keeps the user on the origi- nating page. **206 Partial Content** This code is like 200, except that it is returned by servers in response to range requests. The browser must already have a portion of the document (or it would not have issued a range request) and will normally inspect the *Content-Range* response header to reassem- ble the document before further processing it.

300–399: Redirection and Other Status Messages These codes are used to communicate a variety of states that do not indicate an error but that require special handling on the browser end:
301 Moved Permanently, 302 Found, 303 See Other This response instructs the browser to retry the request at a new location, specified in the *Location* response header. Despite the distinctions made in the RFC, when

encountering any of these response codes, all modern browsers replace POST with GET, remove the payload, and then resubmit the request automatically.

NOTE *Redirect messages may contain a payload, but if they do, this message will not be shown to the user unless the redirection is not possible (for example, because of a missing or unsupported* Location *value). In fact, in some browsers, display of the message may be suppressed even in that scenario.*

304 Not Modified This nonredirect response instructs the client that the requested document hasn't been modified in relation to the copy the client already has. This response is seen after conditional requests with headers such as *If-Modified-Since*, which are issued to revalidate the browser document cache. The response body is not shown to the user. (If the server responds this way to an unconditional request, the result will be browser-specific and may be hilarious; for example, Opera will pop up a nonfunctional download prompt.) **307 Temporary Redirect** Similar to 302, but unlike with other modes of redirection, browsers will not downgrade POST to GET when follow- ing a 307 redirect. This code is not commonly used in web applications, and some browsers do not behave very consistently when handling it.

400–499: Client-Side Error

This range of codes is used to indicate error conditions caused by the behav- ior of the client:

400 Bad Request (and related messages) The server is unable or unwill- ing to process the request for some unspecified reason. The response pay- load will usually explain the problem to some extent and will be typically handled by the browser just like a 200 response.

More specific variants, such as "411 Length Required," "405 Method Not Allowed," or "414 Request-URI Too Long," also exist. It's anyone's guess as to why not specifying *Content-Length* when required has a dedi- cated 411 response code but not specifying *Host* deserves only a generic 400 one. **401 Unauthorized** This code means that the user needs to provide protocol-level HTTP authentication credentials in order to access the resource. The browser will usually prompt the user for login information next, and it will present a response body only if the authentication pro- cess is unsuccessful. This mechanism will be explained in more detail shortly, in "HTTP Authentication" on page 62.

403 Forbidden The requested URL exists but can't be accessed for reasons other than incorrect HTTP authentication. Reasons may involve insufficient filesystem permissions, a configuration rule that prevents this request from being processed, or insufficient credentials of some sort (e.g., invalid cookies or an unrecognized source IP address). The response will usually be shown to the user. **404 Not Found** The requested URL does not exist. The response body is typically shown to the user.

500–599: Server-Side Error

This is a class of error messages returned in response to server-side problems:

500 Internal Server Error, 503 Service Unavailable, and so on The server is experiencing a problem that prevents it from fulfilling the request. This may be a transient condition, a result of misconfiguration, or simply the effect of requesting an unexpected location. The response is normally shown to the user.

Consistency of HTTP Code Signaling

Because there is no immediately observable difference between returning most 2xx, 4xx, and 5xx codes, these values are not selected with any special zeal. In particular, web applications are notorious for returning "200 OK" even when an application error has occurred and is communicated on the resulting page. (This is one of the many factors that make automated testing of web applications much harder than it needs to be.)

On rare occasions, new and not necessarily appropriate HTTP codes are invented for specific uses. Some of these are standardized, such as a couple of messages introduced in the WebDAV RFC.[13] Others, such as Microsoft's Microsoft Exchange "449 Retry With" status, are not.

Keepalive Sessions

Originally, HTTP sessions were meant to happen in one shot: Make one request for each TCP connection, rinse, and

repeat. The overhead of repeat- edly completing a three-step TCP handshake (and forking off a new process in the traditional Unix server design model) soon proved to be a bottleneck, so HTTP/1.1 standardized the idea of keepalive sessions instead.
The existing protocol already gave the server an understanding of where the client request ended (an empty line, optionally followed by *Content-Length* bytes of data), but to continue using the existing connection, the client also needed to know the same about the returned document; the termination of a connection could no longer serve as an indicator. Therefore, keepalive ses- sions require the response to include a *Content-Length* header too, always speci- fying the amount of data to follow. Once this many payload bytes are received, the client knows it is okay to send a second request and begin waiting for another response.
Although very beneficial from a performance standpoint, the way this mechanism is designed exacerbates the impact of HTTP request and response- splitting bugs. It is deceptively easy for the client and the server to get out of sync on which response belongs to which request. To illustrate, let's consider a server that thinks it is sending a single HTTP response, structured as follows:

```
HTTP/1.1 200 OK[CR][LF] Set-Cookie: term=[CR]Content-Length: 0[CR][CR]HTTP/1.1 200 OK[CR]Gotcha:
Yup[CR][LF] Content-Length: 17[CR][LF] [CR][LF] Action completed.
```

The client, on the other hand, may see two responses and associate the first one with its most current request and the second one with the yet-to-be- issued query[*] (which may even be addressed to a different hostname on the same IP):

```
HTTP/1.1 200 OK Set-Cookie: term= Content-Length: 0
HTTP/1.1 200 OK Gotcha: Yup Content-Length: 17
Action completed.
```

If this response is seen by a caching HTTP proxy, the incorrect result may also be cached globally and returned to other users, which is really bad news. A much safer design for keepalive sessions would involve specifying the length of both the headers and the payload up front or using a randomly gen- erated and unpredictable boundary to delimit every response. Regrettably, the design does neither.
Keepalive connections are the default in HTTP/1.1 unless they are explicitly turned off (*Connection: close*) and are supported by many HTTP/1.0 servers when enabled with a *Connection: keep-alive* header. Both servers and browsers can limit the number of concurrent requests serviced per connec- tion and can specify the maximum amount of time an idle connection is kept around.

Chunked Data Transfers

The significant limitation of *Content-Length*-based keepalive sessions is the need for the server to know in advance the exact size of the returned response. This is a pretty simple task when dealing with static files, as the

[*] In principle, clients could be designed to sink any unsolicited server response data before issuing any subsequent requests in a keepalive session, limiting the impact of the attack. This proposal is undermined by the practice of HTTP pipelining, however; for performance reasons, some clients are designed to dump multiple requests at once, without waiting for a complete response in between.

information is already available in the filesystem. When serving dynamically generated data, the problem is more complicated, as the output must be cached in its entirety before it is sent to the client. The challenge becomes insurmountable if the payload is very large or is produced gradually (think live video streaming). In these cases, precaching to compute payload size is simply out of the question.
In response to this challenge, RFC 2616 section 3.6.1 gives servers the ability to use *Transfer-Encoding: chunked*, a scheme in which the payload is sent in portions as it becomes available. The length of every portion of the docu- ment is declared up front using a hexadecimal integer occupying a separate line, but the total length of the document is indeterminate until a final zero- length chunk is seen.
A sample chunked response may look like this:

```
HTTP/1.1 200 OK Transfer-Encoding: chunked ...
```

^{5}Hello ^{6}world! ^{0}

There are no significant downsides to supporting chunked data trans- fers, other than the possibility of pathologically large chunks causing integer overflows in the browser code or needing to resolve mismatches between *Content-Length* and chunk length. (The specification gives precedence to chunk length, although any attempts to handle this situation gracefully appear to be ill-advised.) All the popular browsers deal with these conditions prop- erly, but new implementations need to watch their backs.

Caching Behavior

For reasons of performance and bandwidth conservation, HTTP clients and some intermediaries are eager to cache HTTP responses for later reuse. This must have seemed like a simple task in the early days of the Web, but it is increasingly fraught with peril as the Web encompasses ever more sensi- tive, user-specific information and as this information is updated more and more frequently.

RFC 2616 section 13.4 states that GET requests responded to with a range of HTTP codes (most notably, “200 OK” and “301 Moved Permanently”) may be implicitly cached in the absence of any other server-provided directives. Such a response may be stored in the cache indefinitely, and may be reused for any future requests involving the same request method and destination URL, even if other parameters (such as *Cookie* headers) differ. There is a pro- hibition against caching requests that use HTTP authentication (see “HTTP Authentication” on page 62), but other authentication methods, such as cookies, are not recognized in the spec.