The Tangled Web

A Guide to Securing Modern Web Applications

Michal Zalewski

no starch press

PRAISE FOR *THE TANGLED WEB*

"Thorough and comprehensive coverage from one of the foremost experts in browser security."
—Tavis Ormandy, Google Inc.

"A must-read for anyone who values their security and privacy online."
—Collin Jackson, researcher at the Carnegie Mellon Web Security Group

"Perhaps the most thorough and insightful treatise on the state of security for web-driven technologies to date. A must have!"
—Mark Dowd, Azimuth Security, author of *The Art of Software Security Assessment*

PRAISE FOR *SILENCE ON THE WIRE* BY MICHAL ZALEWSKI

"One of the most innovative and original computing books available."
—Richard Bejtlich, TaoSecurity

"For the pure information security specialist this book is pure gold."
—Mitch Tulloch, Windows Security

"Zalewski's explanations make it clear that he's tops in the industry."
—Computerworld

"The amount of detail is stunning for such a small volume and the examples are amazing. . . . You will definitely think different after reading this title."
—(IN)SECURE Magazine

"Totally rises head and shoulders above other such security-related titles."
—Linux User & Developer

THE TANGLED WEB

A Guide to Securing Modern Web Applications

by Michal Zalewski

no starch
press

San Francisco

THE TANGLED WEB. Copyright © 2012 by Michal Zalewski.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

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
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. "The Book of" is a trademark of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

For my son

BRIEF CONTENTS

Preface xvii

Chapter 1: Security in the World of Web Applications 1

PART I: ANATOMY OF THE WEB 21

Chapter 2: It Starts with a URL 23

Chapter 3: Hypertext Transfer Protocol 41

Chapter 4: Hypertext Markup Language 69

Chapter 5: Cascading Style Sheets 87

Chapter 6: Browser-Side Scripts 95

Chapter 7: Non-HTML Document Types 117

Chapter 8: Content Rendering with Browser Plug-ins 127

PART II: BROWSER SECURITY FEATURES 139

Chapter 9: Content Isolation Logic 141

Chapter 10: Origin Inheritance 165

Chapter 11: Life Outside Same-Origin Rules 173

Chapter 12: Other Security Boundaries 187

Chapter 13: Content Recognition Mechanisms 197

Chapter 14: Dealing with Rogue Scripts 213

Chapter 15: Extrinsic Site Privileges 225

PART III: A GLIMPSE OF THINGS TO COME 233

Chapter 16: New and Upcoming Security Features 235

Chapter 17: Other Browser Mechanisms of Note 255

Chapter 18: Common Web Vulnerabilities 261

Epilogue 267

Notes 269

Index 283

CONTENTS IN DETAIL

PREFACE **xvii**

Acknowledgments xix

1
SECURITY IN THE WORLD OF WEB APPLICATIONS **1**

Information Security in a Nutshell 1
Flirting with Formal Solutions 2
Enter Risk Management 4
Enlightenment Through Taxonomy 6
Toward Practical Approaches 7
A Brief History of the Web 8
Tales of the Stone Age: 1945 to 1994 8
The First Browser Wars: 1995 to 1999 10
The Boring Period: 2000 to 2003 11
Web 2.0 and the Second Browser Wars: 2004 and Beyond 12
The Evolution of a Threat 14
The User as a Security Flaw 14
The Cloud, or the Joys of Communal Living 15
Nonconvergence of Visions 15
Cross-Browser Interactions: Synergy in Failure 16
The Breakdown of the Client-Server Divide 17

PART I: ANATOMY OF THE WEB **21**

2
IT STARTS WITH A URL **23**

Uniform Resource Locator Structure 24
Scheme Name 24
Indicator of a Hierarchical URL 25
Credentials to Access the Resource 26
Server Address 26
Server Port 27
Hierarchical File Path 27
Query String 28
Fragment ID 28
Putting It All Together Again 29
Reserved Characters and Percent Encoding 31
Handling of Non-US-ASCII Text 32
Common URL Schemes and Their Function 36
Browser-Supported, Document-Fetching Protocols 36
Protocols Claimed by Third-Party Applications and Plug-ins 36
Nonencapsulating Pseudo-Protocols 37
Encapsulating Pseudo-Protocols 37
Closing Note on Scheme Detection 38

Resolution of Relative URLs 38
Security Engineering Cheat Sheet 40
When Constructing Brand-New URLs Based on User Input 40
When Designing URL Input Filters 40
When Decoding Parameters Received Through URLs 40

3
HYPERTEXT TRANSFER PROTOCOL 41

Basic Syntax of HTTP Traffic 42
The Consequences of Supporting HTTP/0.9 44
Newline Handling Quirks 45
Proxy Requests 46
Resolution of Duplicate or Conflicting Headers 47
Semicolon-Delimited Header Values 48
Header Character Set and Encoding Schemes 49
Referer Header Behavior 51
HTTP Request Types 52
GET 52
POST 52
HEAD 53
OPTIONS 53
PUT 53
DELETE 53
TRACE 53
CONNECT 54
Other HTTP Methods 54
Server Response Codes 54
200–299: Success 54
300–399: Redirection and Other Status Messages 55
400–499: Client-Side Error 55
500–599: Server-Side Error 56
Consistency of HTTP Code Signaling 56
Keepalive Sessions 56
Chunked Data Transfers 57
Caching Behavior 58
HTTP Cookie Semantics 60
HTTP Authentication 62
Protocol-Level Encryption and Client Certificates 64
Extended Validation Certificates 65
Error-Handling Rules 65
Security Engineering Cheat Sheet 67
When Handling User-Controlled Filenames in Content-Disposition Headers 67
When Putting User Data in HTTP Cookies 67
When Sending User-Controlled Location Headers 67
When Sending User-Controlled Redirect Headers 67
When Constructing Other Types of User-Controlled Requests or Responses 67

4
HYPERTEXT MARKUP LANGUAGE 69

Basic Concepts Behind HTML Documents 70
Document Parsing Modes 71
The Battle over Semantics 72
Understanding HTML Parser Behavior 73
Interactions Between Multiple Tags 74
Explicit and Implicit Conditionals 75
HTML Parsing Survival Tips 76
Entity Encoding 76
HTTP/HTML Integration Semantics 78
Hyperlinking and Content Inclusion 79
Plain Links 79
Forms and Form-Triggered Requests 80
Frames 82
Type-Specific Content Inclusion 82
A Note on Cross-Site Request Forgery 84
Security Engineering Cheat Sheet 85
Good Engineering Hygiene for All HTML Documents 85
When Generating HTML Documents with Attacker-Controlled Bits 85
When Converting HTML to Plaintext 85
When Writing a Markup Filter for User Content 86

5
CASCADING STYLE SHEETS 87

Basic CSS Syntax 88
Property Definitions 89
@ Directives and XBL Bindings 89
Interactions with HTML 90
Parser Resynchronization Risks 90
Character Encoding 91
Security Engineering Cheat Sheet 93
When Loading Remote Stylesheets 93
When Putting Attacker-Controlled Values into CSS 93
When Filtering User-Supplied CSS 93
When Allowing User-Specified Class Values on HTML Markup 93

6
BROWSER-SIDE SCRIPTS 95

Basic Characteristics of JavaScript 96
Script Processing Model 97
Execution Ordering Control 100
Code and Object Inspection Capabilities 101
Modifying the Runtime Environment 102
JavaScript Object Notation and Other Data Serializations 104
E4X and Other Syntax Extensions 106

Standard Object Hierarchy 107
The Document Object Model 109
Access to Other Documents 111
Script Character Encoding 112
Code Inclusion Modes and Nesting Risks 113
The Living Dead: Visual Basic 114
Security Engineering Cheat Sheet 115
When Loading Remote Scripts 115
When Parsing JSON Received from the Server 115
When Putting User-Supplied Data Inside JavaScript Blocks 115
When Interacting with Browser Objects on the Client Side 115
If You Want to Allow User-Controlled Scripts on Your Page 116

7
NON-HTML DOCUMENT TYPES 117

Plaintext Files 117
Bitmap Images 118
Audio and Video 119
XML-Based Documents 119
Generic XML View 120
Scalable Vector Graphics 121
Mathematical Markup Language 122
XML User Interface Language 122
Wireless Markup Language 123
RSS and Atom Feeds 123
A Note on Nonrenderable File Types 124
Security Engineering Cheat Sheet 125
When Hosting XML-Based Document Formats 125
On All Non-HTML Document Types 125

8
CONTENT RENDERING WITH BROWSER PLUG-INS 127

Invoking a Plug-in 128
The Perils of Plug-in Content-Type Handling 129
Document Rendering Helpers 130
Plug-in-Based Application Frameworks 131
Adobe Flash 132
Microsoft Silverlight 134
Sun Java 134
XML Browser Applications (XBAP) 135
ActiveX Controls 136
Living with Other Plug-ins 137
Security Engineering Cheat Sheet 138
When Serving Plug-in-Handled Files 138
When Embedding Plug-in-Handled Files 138
If You Want to Write a New Browser Plug-in or ActiveX Component 138

PART II: BROWSER SECURITY FEATURES 139

9
CONTENT ISOLATION LOGIC 141

Same-Origin Policy for the Document Object Model 142
document.domain 143
postMessage(...) 144
Interactions with Browser Credentials 145
Same-Origin Policy for XMLHttpRequest 146
Same-Origin Policy for Web Storage 148
Security Policy for Cookies 149
Impact of Cookies on the Same-Origin Policy 150
Problems with Domain Restrictions 151
The Unusual Danger of "localhost" 152
Cookies and "Legitimate" DNS Hijacking 153
Plug-in Security Rules 153
Adobe Flash 154
Microsoft Silverlight 157
Java 157
Coping with Ambiguous or Unexpected Origins 158
IP Addresses 158
Hostnames with Extra Periods 159
Non–Fully Qualified Hostnames 159
Local Files 159
Pseudo-URLs 161
Browser Extensions and UI 161
Other Uses of Origins 161
Security Engineering Cheat Sheet 162
Good Security Policy Hygiene for All Websites 162
When Relying on HTTP Cookies for Authentication 162
When Arranging Cross-Domain Communications in JavaScript 162
When Embedding Plug-in-Handled Active Content from Third Parties 162
When Hosting Your Own Plug-in-Executed Content 163
When Writing Browser Extensions 163

10
ORIGIN INHERITANCE 165

Origin Inheritance for about:blank 166
Inheritance for data: URLs 167
Inheritance for javascript: and vbscript: URLs 169
A Note on Restricted Pseudo-URLs 170
Security Engineering Cheat Sheet 172

11
LIFE OUTSIDE SAME-ORIGIN RULES 173

Window and Frame Interactions 174
Changing the Location of Existing Documents 174
Unsolicited Framing 178

Cross-Domain Content Inclusion 181
A Note on Cross-Origin Subresources 183
Privacy-Related Side Channels 184
Other SOP Loopholes and Their Uses 185
Security Engineering Cheat Sheet 186
Good Security Hygiene for All Websites 186
When Including Cross-Domain Resources 186
When Arranging Cross-Domain Communications in JavaScript 186

12
OTHER SECURITY BOUNDARIES 187

Navigation to Sensitive Schemes 188
Access to Internal Networks 189
Prohibited Ports 190
Limitations on Third-Party Cookies 192
Security Engineering Cheat Sheet 195
When Building Web Applications on Internal Networks 195
When Launching Non-HTTP Services, Particularly on Nonstandard Ports 195
When Using Third-Party Cookies for Gadgets or Sandboxed Content 195

13
CONTENT RECOGNITION MECHANISMS 197

Document Type Detection Logic 198
Malformed MIME Types 199
Special Content-Type Values 200
Unrecognized Content Type 202
Defensive Uses of Content-Disposition 203
Content Directives on Subresources 204
Downloaded Files and Other Non-HTTP Content 205
Character Set Handling 206
Byte Order Marks 208
Character Set Inheritance and Override 209
Markup-Controlled Charset on Subresources 209
Detection for Non-HTTP Files 210
Security Engineering Cheat Sheet 212
Good Security Practices for All Websites 212
When Generating Documents with Partly Attacker-Controlled Contents 212
When Hosting User-Generated Files 212

14
DEALING WITH ROGUE SCRIPTS 213

Denial-of-Service Attacks 214
Execution Time and Memory Use Restrictions 215
Connection Limits 216
Pop-Up Filtering 217
Dialog Use Restrictions 218
Window-Positioning and Appearance Problems 219
Timing Attacks on User Interfaces 222

Security Engineering Cheat Sheet 224
When Permitting User-Created <iframe> Gadgets on Your Site 224
When Building Security-Sensitive UIs 224

15
EXTRINSIC SITE PRIVILEGES 225

Browser- and Plug-in-Managed Site Permissions 226
Hardcoded Domains 227
Form-Based Password Managers 227
Internet Explorer's Zone Model 229
Mark of the Web and Zone.Identifier 231
Security Engineering Cheat Sheet 232
When Requesting Elevated Permissions from Within a Web Application 232
When Writing Plug-ins or Extensions That Recognize Privileged Origins 232

PART III: A GLIMPSE OF THINGS TO COME 233

16
NEW AND UPCOMING SECURITY FEATURES 235

Security Model Extension Frameworks 236
Cross-Domain Requests 236
XDomainRequest 239
Other Uses of the Origin Header 240
Security Model Restriction Frameworks 241
Content Security Policy 242
Sandboxed Frames 245
Strict Transport Security 248
Private Browsing Modes 249
Other Developments 250
In-Browser HTML Sanitizers 250
XSS Filtering 251
Security Engineering Cheat Sheet 253

17
OTHER BROWSER MECHANISMS OF NOTE 255

URL- and Protocol-Level Proposals 256
Content-Level Features 258
I/O Interfaces 259

18
COMMON WEB VULNERABILITIES 261

Vulnerabilities Specific to Web Applications 262
Problems to Keep in Mind in Web Application Design 263
Common Problems Unique to Server-Side Code 265

EPILOGUE **267**

NOTES **269**

INDEX **273**

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 dedicated 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 causing 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 security 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 operation. 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 vulnerability 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 journal 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 security 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 prominent, 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 engineering 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 information 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.

1

SECURITY IN THE WORLD OF WEB APPLICATIONS

To provide proper context for the technical discussions 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 treatment. 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 international 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 security measures (such as vulnerability management, malware and attack detection, 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 information 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 algorithmically prove they behave just the right way. This is a simple premise that intuitively 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 computing, it escapes all attempts to usefully define it. Sure, we can restate the problem 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 specificity, 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 security 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 significantly and shift rapidly—when the stakeholders are capable and willing to clearly and honestly disclose their interests to begin with. To add insult to injury, sociology and game theory suggest that computing a simple sum of these particular interests may not actually result in a beneficial 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 system should behave in a subset of cases, it is nearly impossible to formalize 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 particularly 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 processes 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 introduced, all attempts to understand and evaluate the security of real-world software 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 modern 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 according to the following formula:

$$risk = probability\ of\ an\ event \times maximum\ loss$$

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 contingency 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 introduces 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 typical 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 helpfully 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 reliable way of telling how complete it is or how much extra risk one's current 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 lightning 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 acceptable 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, reconstruct, 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 Homeland 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, delightfully 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 significance 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 processes 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 malicious 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 important to implement adequate component separation, access control, data redundancy, monitoring, and response procedures so that service owners 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 engineering 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 contrary, such products are no substitute for street smarts and technical prowess—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 philosophical 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 variety, 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 simple 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 mechanisms 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 tumultuous and unusual way in which the associated technologies have evolved.

So, pardon me another brief detour as we return to the roots. The prehistory 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 simple 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 reference 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 development of HyperText Transfer Protocol (HTTP), an extremely basic, dedicated scheme for accessing HTML resources using the existing concepts of Internet Protocol (IP) addresses, domain names, and file paths. The culmination 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.

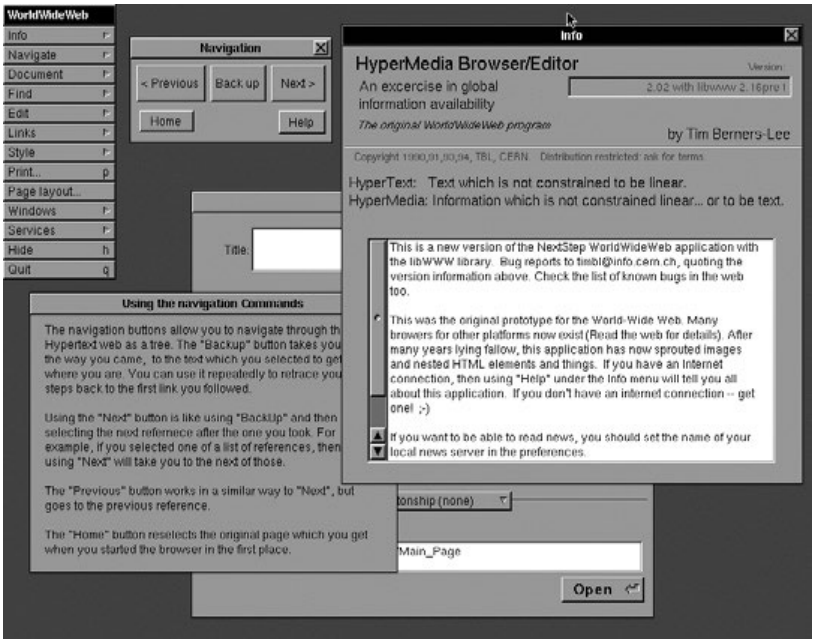


Figure 1-1: Tim Berners-Lee's World Wide Web

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 sophisticated 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 competing products, a trend that often defied all attempts to standardize or even properly 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 inevitably 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 increasingly 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 govern 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 format 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 getting buy-in from the vendors.

Other efforts to standardize or improve already implemented mechanisms, most notably HTTP and JavaScript, were driven by other auspices such as the European Computer Manufacturers Association (ECMA), the International 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 technology. The results were a number of dead standards, contradictory advice, and several frightening examples of harmful cross-interactions between otherwise 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 feature 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 brilliantly, 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 interest, 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 significant 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 disruptive 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 responsive, 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 vulnerabilities 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 offensive, 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 stubborn 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 product, the Windows operating system, was increasingly being used as an (expendable?) 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 improvements. Caught off guard by the unexpected success of *XMLHttpRequest* and quick to forget other lessons from the past, vendors also decided to experiment 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 reportedly 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 communication 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 particularly 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 remarkably 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 without 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 operating 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 interaction 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 security mechanisms offer rudimentary protection against the most obvious attacks. But this brings us to another characteristic that makes the Web such an interesting 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 permission 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 realizes 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 (essentially 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 difficult 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 software 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 exposure 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 browsers 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 inadequate 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 developers 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 interactions 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. Developers learn these roles and play along, building fairly comprehensible and testable network-computing environments in the process.

The Web began as a classical example of a proper client-server architecture, 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 synchronously 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 leaving 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 severed 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 programming 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.

2

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 conclusively address and uniquely identify a single resource on a remote server (and in doing so, implement a couple 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 common 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 vendors 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.

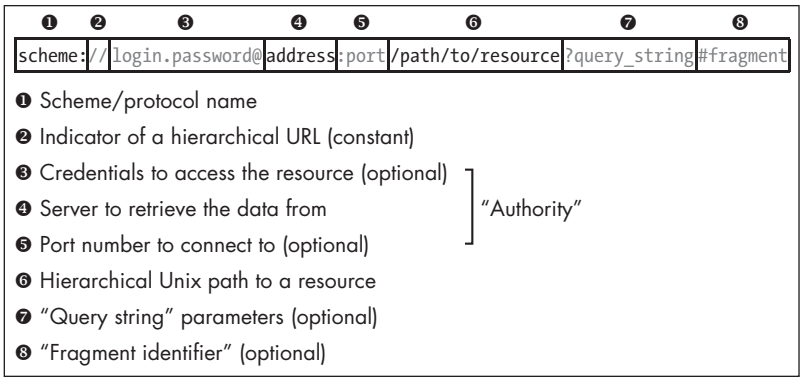


Figure 2-1: Structure of an absolute URL

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 applications, some which have special security consequences. (Of particular interest are several types of pseudo-URLs, such as *data:* or *javascript:*, as discussed later in this chapter and throughout the remainder of this book.)

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 practice, 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 ability 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 specification 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 elegant 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 counterintuitive meaning to the latter class of URLs. In RFC 3986, published some years later, the authors sheepishly acknowledge this flaw and permit implementations to try to parse such URLs for compatibility reasons. As a consequence, 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 behavior 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 reasons, 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 specify 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, standard C libraries used by most applications are much more relaxed, accepting noncanonical IPv4 addresses that mix octal, decimal, and hexadecimal notation 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, alphanumerics, ".", 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 characters 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 reportedly 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 contemporary 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, nonhierarchical 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 whatever 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 document 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 portion 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 reason 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 browsers, "\" 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 username, 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 legitimate 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 compelled 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 inconsistencies. 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. Therefore, 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 browsers 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 browsers 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. Unicode 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 "±", "ą", 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 contained a literal non-US-ASCII character anywhere in the hostname was supposed 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.xn--rczniki-98a.pl/r%C4%99cznik?model=Ja%B6#Złóż_zamówienie
```

Label converted to Punycode

Path converted to UTF-8

Query string converted to ISO 8859-2

Literal UTF-8

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 *example.com* domain and an impostor *examp1e.com* site are the same thing. But IDNA casually and indiscriminately extended these 38 characters to some 100,000 glyphs supported by Unicode, many of which look exactly alike and are separated from each other based on functional differences alone.

How bad is it? Let's consider Cyrillic, for example. This alphabet has a number of homoglyphs that look practically identical to their Latin counterparts but that have completely different Unicode values and resolve to completely 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	a	c	e	i	j	o	p	s	x	y
	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 apparently 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 published "The Homograph Attack,"[11] a paper exploring the vulnerability in great detail. They noted that any registrar-level work-arounds, even if implemented, would have a fatal flaw. An attacker could always purchase a wholesome 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 nonfunctional 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 envisioned 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 certain country-specific, top-level domains (ruling out the use of internationalized 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 application 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 browsers 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, specialized 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 software; *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 sometimes 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 protocols 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 Internet 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 browsers). 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 appearance 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 supported 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 subsystem 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 applications 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 chapter, 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 considerable 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 scenarios 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, preferably, 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 sections, 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 classified 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 authority 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 circumstances, as noted earlier.

Because of the risk of potential misunderstandings between application-level URL filters and the browser when handling these types of relative references, it is a good design practice never to output user-supplied relative URLs verbatim. Where feasible, they should be explicitly rewritten to absolute references, 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 different 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 injection 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 numbers, or HTTP credentials in the URL. If so, you must fully parse the URL, validate all sections 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 putting them inside database queries, new URLs, and so on, scrub them carefully for dangerous characters.

3

HYPERTEXT TRANSFER PROTOCOL

The next essential concept we need to discuss is the Hypertext Transfer Protocol (HTTP): the core transfer 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 extensibility 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 shortcomings 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 anticipate 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 practical need to do so, several web servers, and all common browsers, also maintain 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 preferences, 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 reasonably 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 administrators: 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 protocol 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 information), *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 hidden 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 servers 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 illustrate 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 continuation 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 authenticated 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 mismatching *Host* headers to match the request URL or associate cached content 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 protocol, 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 receiving 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 communicate 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 dangerous: 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 clients and servers can have a field day with this behavior. Without these additional 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 "unambiguous" cases, but the meaning of the term itself is, shall we say, not particularly 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 injection vulnerability, no matter how constrained, is exploitable for at least some percentage of targeted users. On the server side, the situation is similarly random: 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 conflicts 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 proxies!) 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 implementations follow this advice. The problem is that underlying applications are likely to be unaware of this quirk and may instead make somewhat important decisions 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 confusing 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 otherwise 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 document) 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 characters) 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 incompatible 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 reloading 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 subsequent 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, providing 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 characters 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 "=?" prefix, 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 cooperative. 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 supported by all browsers at once. This situation prompts some web application developers to resort to using raw high-bit values in the HTTP headers, typically 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 alphabet 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 materialize 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, dysfunctional *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 truncated 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 current 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 information 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 purposes, 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 settings or installing a privacy-oriented plug-in.

As should be apparent, four out of five of these conditions can be purposefully 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 modern 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 application 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 overlapping 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 persistent 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 format 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 cornerstone 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 upcoming 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 servers in a remote network. Precisely for this reason, the method is often disabled 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 applications; the only reason for their existence is that somebody hoped they eventually 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 originating 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 reassemble 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 following 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 behavior of the client:

400 Bad Request (and related messages) The server is unable or unwilling to process the request for some unspecified reason. The response payload 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 dedicated 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 process 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 repeatedly 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 sessions require the response to include a *Content-Length* header too, always specifying 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 generated 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 connection 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 document 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 transfers, 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 properly, 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 sensitive, 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 prohibition 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.

When a response is cached, the implementation may opt to revalidate it before reuse, but doing so is not required most of the time. Revalidation is achieved by request with a special conditional header, such as *If-Modified-Since* (followed by a date recorded on the previously cached response) or *If-None-Match* (followed by an opaque *ETag* header value that the server returned with an earlier copy). The server may respond with a "304 Not Modified" code or return a newer copy of the resource.

NOTE *The* Date/If-Modified-Since *and* ETag/If-None-Match *header pairs, when coupled with* Cache-Control: private*, offer a convenient and entirely unintended way for websites to store long-lived, unique tokens in the browser.*[14] *The same can also be achieved by depositing a unique token inside a cacheable JavaScript file and returning "304 Not Modified" to all future conditional requests to the token-generating location. Unlike purpose-built mechanisms such as HTTP cookies (discussed in the next section), users have very little control over what information is stored in the browser cache, under what circumstances, and for how long.*

Implicit caching is highly problematic, and therefore, servers almost always should resort to using explicit HTTP-caching directives. To assist with this, HTTP/1.0 provides an *Expires* header that specifies the date by which the cached copy should be discarded; if this value is equal to the *Date* header provided by the server, the response is noncacheable. Beyond that simple rule, the connection between *Expires* and *Date* is unspecified: It is not clear whether *Expires* should be compared to the system clock on the caching system (which is problematic if the client and server clocks are not in sync) or evaluated based on the *Expires – Date* delta (which is more robust, but which may stop working if *Date* is accidentally omitted). Firefox and Opera use the latter interpretation, while other browsers prefer the former one. In most browsers, an invalid *Expires* value also inhibits caching, but depending on it is a risky bet.

HTTP/1.0 clients can also include a *Pragma: no-cache* request header, which may be interpreted by the proxy as an instruction to obtain a new copy of the requested resource, instead of returning an existing one. Some HTTP/1.0 proxies also recognize a nonstandard *Pragma: no-cache* response header as an instruction not to make a copy of the document.

In contrast, HTTP/1.1 embraces a far more substantial approach to caching directives, introducing a new *Cache-Control* header. The header takes values such as *public* (the document is cacheable publicly), *private* (proxies are not permitted to cache), *no-cache* (which is a bit confusing—the response may be cached but should not be reused for future requests),* and *no-store* (absolutely no caching at all). Public and private caching directives may be accompanied with a qualifier such as *max-age*, specifying the maximum time an old copy should be kept, or *must-revalidate*, requesting a conditional request to be made before content reuse.

* The RFC is a bit hazy in this regard, but it appears that the intent is to permit the cached document to be used for purposes such as operating the "back" and "forward" navigation buttons in a browser but not when a proper page load is requested. Firefox follows this approach, while all other browsers consider *no-cache* and *no-store* to be roughly equivalent.

Unfortunately, it is typically necessary for servers to return both HTTP/1.0 and HTTP/1.1 caching directives, because certain types of legacy commercial proxies do not understand *Cache-Control* correctly. In order to reliably prevent caching over HTTP, it may be necessary to use the following set of response headers:

```
Expires: [current date]
Date: [current date]
Pragma: no-cache
Cache-Control: no-cache, no-store
```

When these caching directives disagree, the behavior is difficult to predict: Some browsers will favor HTTP/1.1 directives and give precedence to *no-cache*, even if it is mistakenly followed by *public*; others don't.

Another risk of HTTP caching is associated with unsafe networks, such as public Wi-Fi networks, which allow an attacker to intercept requests to certain URLs and return modified, long-cacheable contents on requests to the victim. If such a poisoned browser cache is then reused on a trusted network, the injected content will unexpectedly resurface. Perversely, the victim does not even have to visit the targeted application: A reference to a carefully chosen sensitive domain can be injected by the attacker into some other context. There are no good solutions to this problem yet; purging your browser cache after visiting Starbucks may be a very good idea.

HTTP Cookie Semantics

HTTP cookies are not a part of RFC 2616, but they are one of the more important protocol extensions used on the Web. The cookie mechanism allows servers to store short, opaque *name=value* pairs in the browser by sending a *Set-Cookie* response header and to receive them back on future requests via the client-supplied *Cookie* parameter. Cookies are by far the most popular way to maintain sessions and authenticate user requests; they are one of the four canonical forms of *ambient authority*[*] on the Web (the other forms being built-in HTTP authentication, IP checking, and client certificates).

Originally implemented in Netscape by Lou Montulli around 1994, and described in a brief four-page draft document,[15] the mechanism has not been outlined in a proper standard in the last 17 years. In 1997, RFC 2109[16] attempted to document the status quo, but somewhat inexplicably, it also proposed a number of sweeping changes that, to this day, make this specification substantially incompatible with the actual behavior of any modern browser. Another ambitious effort—*Cookie2*—made an appearance in RFC 2965,[17] but a decade later, it still has virtually no browser-level support, a situation that is

[*] *Ambient authority* is a form of access control based on a global and persistent property of the requesting entity, rather than any explicit form of authorization that would be valid only for a specific action. A user-identifying cookie included indiscriminately on every outgoing request to a remote site, without any consideration for why this request is being made, falls into that category.

unlikely to change. A new effort to write a reasonably accurate cookie specification—RFC 6265[18]—was wrapped up shortly before the publication of this book, finally ending this specification-related misery.

Because of the prolonged absence of any real standards, the actual implementations evolved in very interesting and sometimes incompatible ways. In practice, new cookies can be set using *Set-Cookie* headers followed by a single *name=value* pair and a number of optional semicolon-delimited parameters defining the scope and lifetime of the cookie.

Expires Specifies the expiration date for a cookie in a format similar to that used for *Date* or *Expires* HTTP headers. If a cookie is served without an explicit expiration date, it is typically kept in memory for the duration of a browser session (which, especially on portable computers with suspend functionality, can easily span several weeks). Definite-expiry cookies may be routinely saved to disk and persist across sessions, unless a user's privacy settings explicitly prevent this possibility.

Max-age This alternative, RFC-suggested expiration mechanism is not supported in Internet Explorer and therefore is not used in practice.

Domain This parameter allows the cookie to be scoped to a domain broader than the hostname that returned the *Set-Cookie* header. The exact rules and security consequences of this scoping mechanism are explored in Chapter 9.

NOTE *Contrary to what is implied in RFC 2109, it is not possible to scope cookies to a specific hostname when using this parameter. For example,* domain=example.com *will always match* www.example.com *as well. Omitting* domain *is the only way to create host-scoped cookies, but even this approach is not working as expected in Internet Explorer.*

Path Allows the cookie to be scoped to a particular request path prefix. This is not a viable security mechanism for the reasons explained in Chapter 9, but it may be used for convenience, to prevent identically named cookies used in various parts of the application from colliding with each other.

Secure attribute Prevents the resulting cookie from being sent over nonencrypted connections.

HttpOnly attribute Removes the ability to read the cookie through the *document.cookie* API in JavaScript. This is a Microsoft extension, although it is now supported by all mainstream browsers.

When making future requests to a domain for which valid cookies are found in the cookie jar, browsers will combine all applicable *name=value* pairs into a single, semicolon-delimited *Cookie* header, without any additional metadata, and return them to the server. If too many cookies need to be sent on a particular request, server-enforced header size limits will be exceeded, and the request may fail; there is no method for recovering from this condition, other than manually purging the cookie jar.

Curiously, there is no explicit method for HTTP servers to delete unneeded cookies. However, every cookie is uniquely identified by a name-domain-path tuple (the *secure* and *httponly* attributes are ignored), which permits an old cookie of a known scope to be simply overwritten. Furthermore, if the overwriting cookie has an *expires* date in the past, it will be immediately dropped, effectively giving a contrived way to purge the data.

Although RFC 2109 requires multiple comma-separated cookies to be accepted within a single *Set-Cookie* header, this approach is dangerous and is no longer supported by any browser. Firefox allows multiple cookies to be set in a single step via the *document.cookie* JavaScript API, but inexplicably, it requires newlines as delimiters instead. No browser uses commas as *Cookie* delimiters, and recognizing them on the server side should be considered unsafe.

Another important difference between the spec and reality is that cookie values are supposed to use the *quoted-string* format outlined in HTTP specs (see "Semicolon-Delimited Header Values" on page 48), but only Firefox and Opera recognize this syntax in practice. Reliance on *quoted-string* values is therefore unsafe, and so is allowing stray quote characters in attacker-controlled cookies.

Cookies are not guaranteed to be particularly reliable. User agents enforce modest settings on the number and size of cookies permitted per domain and, as a misguided privacy feature, may also restrict their lifetime. Because equally reliable user tracking may be achieved by other means, such as the *ETag/If-None-Match* behavior outlined in the previous section, the efforts to restrict cookie-based tracking probably do more harm than good.

HTTP Authentication

HTTP authentication, as specified in RFC 2617,[19] is the original credential-handling mechanism envisioned for web applications, one that is now almost completely extinct. The reasons for this outcome might have been the inflexibility of the associated browser-level UIs, the difficulty of accommodating more sophisticated non-password-based authentication schemes, or perhaps the inability to exercise control over how long credentials are cached and what other domains they are shared with.

In any case, the basic scheme is fairly simple. It begins with the browser making an unauthenticated request, to which the server responds with a "401 Unauthorized" code.[*] The server must also include a *WWW-Authenticate* HTTP header, specifying the requested authentication method, the *realm* string (an arbitrary identifier to which the entered credentials should be bound), and other method-specific parameters, if applicable.

[*] The terms *authentication* and *authorization* appear to be used interchangeably in this RFC, but they have a distinctive meaning elsewhere in information security. *Authentication* is commonly used to refer to the process of proving your identity, whereas *authorization* is the process of determining whether your previously established credentials permit you to carry out a specific privileged action.

The client is expected to obtain the credentials in one way or the other, encode them in the *Authorization* header, and retry the original request with this header included. According to the specification, for performance reasons, the same *Authorization* header may also be included on subsequent requests to the same server path prefix without the need for a second *WWW-Authenticate* challenge. It is also permissible to reuse the same credentials in response to any *WWW-Authenticate* challenges elsewhere on the server, if the *realm* string and the authentication method match.

In practice, this advice is not followed very closely: Other than Safari and Chrome, most browsers ignore the *realm* string or take a relaxed approach to path matching. On the flip side, all browsers scope cached credentials not only to the destination server but also to a specific protocol and port, a practice that offers some security benefits.

The two credential-passing methods specified in the original RFC are known as *basic* and *digest*. The first one essentially sends the passwords in plaintext, encoded as *base64*. The other computes a one-time cryptographic hash that protects the password from being viewed in plaintext and prevents the *Authorization* header from being replayed later. Unfortunately, modern browsers support both methods and do not distinguish between them in any clear way. As a result, attackers can simply replace the word *digest* with *basic* in the initial request to obtain a clean, plaintext password as soon as the user completes the authentication dialog. Surprisingly, section 4.8 of the RFC predicted this risk and offered some helpful yet ultimately ignored advice:

> User agents should consider measures such as presenting a visual indication at the time of the credentials request of what authentication scheme is to be used, or remembering the strongest authentication scheme ever requested by a server and produce a warning message before using a weaker one. It might also be a good idea for the user agent to be configured to demand Digest authentication in general, or from specific sites.

In addition to these two RFC-specified authentication schemes, some browsers also support less-common methods, such as Microsoft's *NTLM* and *Negotiate*, used for seamless authentication with Windows domain credentials.[20]

Although HTTP authentication is seldom encountered on the Internet, it still casts a long shadow over certain types of web applications. For example, when an external, attacker-supplied image is included in a thread on a message board, and the server hosting that image suddenly decides to return "401 Unauthorized" on some requests, users viewing the thread will be presented out of the blue with a somewhat cryptic password prompt. After double-checking the address bar, many will probably confuse the prompt for a request to enter their forum credentials, and these will be immediately relayed to the attacker's image-hosting server. Oops.

Protocol-Level Encryption and Client Certificates

As should now be evident, all information in HTTP sessions is exchanged in plaintext over the network. In the 1990s, this would not have been a big deal: Sure, plaintext exposed your browsing choices to nosy ISPs, and perhaps to another naughty user on your office network or an overzealous government agency, but that seemed no worse than the behavior of SMTP, DNS, or any other commonly used application protocol. Alas, the growing popularity of the Web as a commerce platform has aggravated the risk, and substantial network security regression caused by the emergence of inherently unsafe public wireless networks put another nail in that coffin.

After several less successful hacks, a straightforward solution to this problem was proposed in RFC 2818:[21] Why not encapsulate normal HTTP requests within an existing, multipurpose Transport Layer Security (TLS, aka SSL) mechanism developed several years earlier? This transport method leverages public key cryptography* to establish a confidential, authenticated communication channel between the two endpoints, without requiring any HTTP-level tweaks.

In order to allow web servers to prove their identity, every HTTPS-enabled web browser ships with a hefty set of public keys belonging to a variety of *certificate authorities*. Certificate authorities are organizations that are trusted by browser vendors to cryptographically attest that a particular public key belongs to a particular site, hopefully after validating the identity of the person who requests such attestation and after verifying his claim to the domain in question.

The set of trusted organizations is diverse, arbitrary, and not particularly well documented, which often prompts valid criticisms. But in the end, the system usually does the job reasonably well. Only a handful of bloopers have been documented so far (including a recent high-profile compromise of a company named Comodo[22]), and no cases of widespread abuse of CA privileges are on the record.

As to the actual implementation, when establishing a new HTTPS connection, the browser receives a signed public key from the server, verifies the signature (which can't be forged without having access to the CA's private key), checks that the signed *cn* (common name) or *subjectAltName* fields in the certificate indicate that this certificate is issued for the server the browser wants to talk to, and confirms that the key is not listed on a public revocation list (for example, due to being compromised or obtained fraudulently). If everything checks out, the browser can proceed by encrypting messages to the server with that public key and be certain that only that specific party will be able to decrypt them.

Normally, the client remains anonymous: It generates a temporary encryption key, but that process does not prove the client's identity. Such a proof can be arranged, though. Client certificates are embraced internally by certain organizations and are adopted on a national level in several countries

* Public key cryptography relies on asymmetrical encryption algorithms to create a pair of keys: a private one, kept secret by the owner and required to decrypt messages, and a public one, broadcast to the world and useful only to encrypt traffic to that recipient, not to decrypt it.

around the world (e.g., for e-government services). Since the usual purpose of a client certificate is to provide some information about the real-world identity of the user, browsers usually prompt before sending them to newly encountered sites, for privacy reasons; beyond that, the certificate may act as yet another form of ambient authority.

It is worth noting that although HTTPS as such is a sound scheme that resists both passive and active attackers, it does very little to hide the evidence of access to a priori public information. It does not mask the rough HTTP request and response sizes, traffic directions, and timing patterns in a typical browsing session, thus making it possible for unsophisticated, passive attackers to figure out, for example, which embarrassing page on Wikipedia is being viewed by the victim over an encrypted channel. In fact, in one extreme case, Microsoft researchers illustrated the use of such packet profiling to reconstruct user keystrokes in an online application.[23]

Extended Validation Certificates

In the early days of HTTPS, many public certificate authorities relied on fairly pedantic and cumbersome user identity and domain ownership checks before they would sign a certificate. Unfortunately, in pursuit of convenience and in the interest of lowering prices, some now require little more than a valid credit card and the ability to put a file on the destination server in order to complete the verification process. This approach renders most of the certificate fields other than *cn* and *subjectAltName* untrustworthy.

To address this problem, a new type of certificate, tagged using a special flag, is being marketed today at a significantly higher price: *Extended Validation SSL (EV SSL)*. These certificates are expected not only to prove domain ownership but also more reliably attest to the identity of the requesting party, following a manual verification process. EV SSL is recognized by all modern browsers by making portion of the address bar blue or green. Although having this tier of certificates is valuable, the idea of coupling a higher-priced certificate with an indicator that vaguely implies a "higher level of security" is often criticized as a cleverly disguised money-making scheme.

Error-Handling Rules

In an ideal world, HTTPS connections that involve a suspicious certificate error, such as a grossly mismatched hostname or an unrecognized certification authority, should simply result in a failure to establish the connection. Less-suspicious errors, such as a recently expired certificate or a hostname mismatch, perhaps could be accompanied by just a gentle warning.

Unfortunately, most browsers have indiscriminately delegated the responsibility for understanding the problem to the user, trying hard (and ultimately failing) to explain cryptography in layman's terms and requiring the user to make a binary decision: Do you actually want to see this page or not? (Figure 3-1 shows one such prompt.)

Figure 3-1: An example certificate warning dialog in the still-popular Internet Explorer 6

The language and appearance of SSL warnings has evolved through the years toward increasingly dumbed-down (but still problematic) explanations of the problem and more complicated actions required to bypass the warning. This trend may be misguided: Studies show that over 50 percent of even the most frightening and disruptive warnings are clicked through.[24] It is easy to blame the users, but ultimately, we may be asking them the wrong questions and offering exactly the wrong choices. Simply, if it is believed that clicking through the warning is advantageous in some cases, offering to open the page in a clearly labeled "sandbox" mode, where the harm is limited, would be a more sensible solution. And if there is no such belief, any override capabilities should be eliminated entirely (a goal sought by *Strict Transport Security*, an experimental mechanism that will be discussed in Chapter 16).

Security Engineering Cheat Sheet

When Handling User-Controlled Filenames in Content-Disposition Headers

☑ **If you do not need non-Latin characters:** Strip or substitute any characters except for alphanumerics, ".", "-", and "_". To protect your users against potentially harmful or deceptive filenames, you may also want to confirm that at least the first character is alphanumeric and substitute all but the rightmost period with something else (e.g., an underscore).

Keep in mind that allowing quotes, semicolons, backslashes, and control characters (0x00–0x1F) will introduce vulnerabilities.

☑ **If you need non-Latin names:** You must use RFC 2047, RFC 2231, or URL-style percent encoding in a browser-dependent manner. Make sure to filter out control characters (0x00–0x1F) and escape any semicolons, backslashes, and quotes.

When Putting User Data in HTTP Cookies

☑ **Percent-encode everything except for alphanumerics.** Better yet, use base64. Stray quote characters, control characters (0x00–0x1F), high-bit characters (0x80–0xFF), commas, semicolons, and backslashes may allow new cookie values to be injected or the meaning and scope of existing cookies to be altered.

When Sending User-Controlled Location Headers

☑ **Consult the cheat sheet in Chapter 2.** Parse and normalize the URL, and confirm that the scheme is on a whitelist of permissible values and that you are comfortable redirecting to the specified host.

Make sure that any control and high-bit characters are escaped properly. Use Punycode for hostnames and percent-encoding for the remainder of the URL.

When Sending User-Controlled Redirect Headers

☑ **Follow the advice provided for Location.** Note that semicolons are unsafe in this header and cannot be escaped reliably, but they also happen to have a special meaning in some URLs. Your choice is to reject such URLs altogether or to percent-encode the ";" character, thereby violating the RFC-mandated syntax rules.

When Constructing Other Types of User-Controlled Requests or Responses

☑ **Examine the syntax and potential side effects of the header in question.** In general, be mindful of control and high-bit characters, commas, quotes, backslashes, and semicolons; other characters or strings may be of concern on a case-by-case basis. Escape or substitute these values as appropriate.

☑ **When building a new HTTP client, server, or proxy:** Do not create a new implementation unless you absolutely have to. If you can't help it, read this chapter thoroughly and aim to mimic an existing mainstream implementation closely. If possible, ignore the RFC-provided advice about fault tolerance and bail out if you encounter any syntax ambiguities.

4

HYPERTEXT MARKUP LANGUAGE

The Hypertext Markup Language (HTML) is the primary method of authoring online documents. One of the earliest written accounts of this language is a brief summary posted on the Internet by Tim Berners-Lee in 1991.[1] His proposal outlines an SGML-derived syntax that allows text documents to be annotated with inline hyperlinks and several types of layout aids. In the following years, this specification evolved gradually under the direction of Sir Berners-Lee and Dan Connolly, but it wasn't until 1995, at the onset of the First Browser Wars, that a reasonably serious and exhaustive specification of the language (HTML 2.0) made it to RFC 1866.[2]

From that point on, all hell broke loose: For the next few years, competing browser vendors kept introducing all sorts of flashy, presentation-oriented features and tweaked the language to their liking. Several attempts to amend the original RFC have been undertaken, but ultimately the IETF-managed

standardization approach proved to be too inflexible. The newly formed World Wide Web Consortium took over the maintenance of the language and eventually published the HTML 3.2 specification in 1997.[3]

The new specification tried to reconcile the differences in browser implementations while embracing many of the bells and whistles that appealed to the public, such as customizable text colors and variable typefaces. Ultimately, though, HTML 3.2 proved to be a step back for the clarity of the language and had only limited success in catching up with the facts.

In the following years, the work on HTML 4 and 4.01[4] focused on pruning HTML of all accumulated excess and on better explaining how document elements should be interpreted and rendered. It also defined an alternative, strict XHTML syntax derived from XML, which was much easier to consistently parse but more punishing to write. Despite all this work, however, only a small fraction of all websites on the Internet could genuinely claim compliance with any of these standards, and little or no consistency in parsing modes and error recovery could be seen on the client end. Consequently, some of the work on improving the core language fizzled out, and the W3C turned its attention to stylesheets, the Document Object Model, and other more abstract or forward-looking challenges.

In the late 2000s, some of the low-level work has been revived under the banner of HTML5,[5] an ambitious project to normalize almost every aspect of the language syntax and parsing, define all the related APIs, and more closely police browser behavior in general. Time will tell if it will be successful; until then, the language itself, and each of the four leading parsing engines,* come with their own set of frustrating quirks.

Basic Concepts Behind HTML Documents

From a purely theoretical standpoint, HTML relies on a fairly simple syntax: a hierarchical structure of tags, *name=value* tag parameters, and text nodes (forming the actual document body) in between. For example, a simple document with a title, a heading, and a hyperlink may look like this:

```
<html>
  <head>
    <title>Hello world</title>
  </head>
  <body>
    <h1>Welcome to our example page</h1>
    <a href="http://www.example.com/">Click me!</a>
  </body>
</html>
```

* To process HTML documents, Internet Explorer uses the Trident engine (aka MSHTML); Firefox and some derived products use Gecko; Safari, Chrome, and several other browsers use WebKit; and Opera relies on Presto. With the exception of WebKit, a collaborative open source effort maintained by several vendors, these engines are developed largely in-house by their respective browser teams.

This syntax puts some constraints on what may appear inside a parameter value or inside the document body. Five characters—angle brackets, single and double quotes, and an ampersand—are reserved as the building blocks of the HTML markup, and these need to be avoided or escaped in some way when used outside of their intended function. The most important rules are:

- Stray ampersands (&) should never appear in most sections of an HTML document.
- Both types of angle brackets are obviously problematic inside a tag, unless properly quoted.
- The left angle bracket (<) is a hazard inside a text node.
- Quote characters appearing inside a tag can have undesirable effects, depending on their exact location, but are harmless in text nodes.

To allow these characters to appear in problematic locations without causing side effects, an ampersand-based encoding scheme, discussed in "Entity Encoding" on page 76, is provided.

NOTE *Of course, the availability of such an encoding scheme is not a guarantee of its use. The failure to properly filter out or escape reserved characters when displaying user-controlled data is the cause of a range of extremely common and deadly web application security flaws. A particularly well-known example of this is* cross-site scripting (XSS)*, an attack in which malicious, attacker-provided JavaScript code is unintentionally echoed back somewhere in the HTML markup, effectively giving the attacker full control over the appearance and operation of the targeted site.*

Document Parsing Modes

For any HTML document, a top-level *<!DOCTYPE>* directive may be used to instruct the browser to parse the file in a manner that at least superficially conforms to one of the officially defined standards; to a more limited extent, the same signal can be conveyed by the *Content-Type* header, too. Of all the available parsing modes, the most striking difference exists between XHTML and traditional HTML. In the traditional mode, parsers will attempt to recover from most types of syntax errors, including unmatched opening and closing tags. In addition, tag and parameter names will be considered case insensitive, parameter values will not always need to be quoted, and certain types of tags, such as *<img>*, will be closed implicitly. In other words, the following input will be grudgingly tolerated:

```
<hTmL>
  <BODY>
    <IMG src="/hello_world.jpg">
    <a HREF=http://www.example.com/>
      Click me!
    </oops>
</html>
```

The XML mode, on the other hand, is strict: All tags need to be balanced carefully, named using the proper case, and closed explicitly. (The XML-specific self-closing tag syntax, such as *<img />*, is permitted.) In addition, most syntax mistakes, even trivial ones, will result in an error and prevent the document from being displayed at all.

Unlike the regular flavor of HTML, XML-based documents may also elegantly incorporate sections using other XML-compliant markup formats, such as MathML, a mathematical formula markup language. This is done by specifying a different *xmlns* namespace setting for a particular tag, with no need for one-off, language-level hacks.

The last important difference worth mentioning here is that traditional HTML parsing strategies feature a selection of special modes, entered into after certain tags are encountered and exited only when a specific terminator string is seen; everything in between is interpreted as non-HTML text. Some examples of such special tags include *<style>*, *<script>*, *<textarea>*, or *<xmp>*. In practical implementations, these modes are exited only when a literal, case-insensitive match on *</style*, *</script*, or a similar matching value, is made; any other markup inside such a block will not be interpreted as HTML. (Interestingly, there is one officially obsolete tag, *<plaintext>*, that cannot be exited at all; it stays in effect for the remainder of the document.)

In comparison, the XML mode is more predictable. It generally forbids stray "<" and "&" characters inside the document, but it provides a special syntax, starting with "<![CDATA[" and ending with "]]>", as a way to encapsulate any raw text inside an arbitrary tag. For example:

```
<script>
<![CDATA[
  alert('>>> Hello world! <<<');
]]>
</script>
```

The other notable special parsing mode available in both XHTML and normal HTML is a comment block. In XML, it quite simply begins with "<!--" and ends with "-->". In the traditional HTML parser in Firefox versions prior to 4, any occurrence of "--", later followed by ">", is also considered good enough.

The Battle over Semantics

The low-level syntax of the language aside, HTML is also the subject of a fascinating conceptual struggle: a clash between the ideology and the reality of the online world. Tim Berners-Lee always championed the vision of a *semantic web*, an interconnected system of documents in which every functional block, such as a citation, a snippet of code, a mailing address, or a heading, has its meaning explained by an appropriate machine-readable tag (say, *<cite>*, *<code>*, *<address>*, or *<h1>* to *<h6>*).

This approach, he and other proponents argued, would make it easier for machines to crawl, analyze, and index the content in a meaningful way, and in the near future, it would enable computers to reason using the sum of human knowledge. According to this philosophy, the markup language should provide a way to stylize the appearance of a document, but only as an afterthought.

Sir Berners-Lee has never given up on this dream, but in this one regard, the actual usage of HTML proved to be very different from what he wished for. Web developers were quick to pragmatically distill the essence of HTML 3.2 into a handful of presentation-altering but semantically neutral tags, such as *<font>*, *<b>*, and *<pre>*, and saw no reason to explain further the structure of their documents to the browser. W3C attempted to combat this trend but with limited success. Although tags such as *<font>* have been successfully obsoleted and largely abandoned in favor of CSS, this is only because stylesheets offered more powerful and consistent visual controls. With the help of CSS, the developers simply started relying on a soup of semantically agnostic ** and * * tags to build everything from headings to user-clickable buttons, all in a manner completely opaque to any automated content extraction tools.

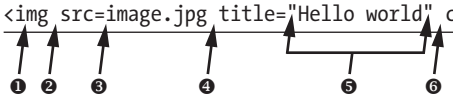
Despite having had a lasting impact on the design of the language, in some ways, the idea of a semantic web may be becoming obsolete: Online content less frequently maps to the concept of a single, viewable document, and HTML is often reduced to providing a convenient drawing surface and graphic primitives for JavaScript applications to build their interfaces with.

Understanding HTML Parser Behavior

The fundamentals of HTML syntax outlined in the previous sections are usually enough to understand the meaning of well-formed HTML and XHTML documents. When the XHTML dialect is used, there is little more to the story: The minimal fault-tolerance of the parser means that anomalous syntax almost always leads simply to a parsing error. Alas, the picture is very different with traditional, laid-back HTML parsers, which aggressively second-guess the intent of the page developer even in very ambiguous or potentially harmful situations.

Since an accurate understanding of user-supplied markup is essential to designing many types of security filters, let's have a quick look at some of these behaviors and quirks. To begin, consider the following reference snippet:

```
<img src=image.jpg title="Hello world" class=examples>
```



Web developers are usually surprised to learn that this syntax can be drastically altered without changing its significance to the browser. For example, Internet Explorer will allow an NUL character (0x00) to be inserted in the location marked at ❶, a change that is likely to throw all naïve HTML filters off the trail. It is also not widely known that the whitespaces at ❷ and ❹ can

be substituted with uncommon vertical tab (0x0B) or form feed (0x0C) characters in all browsers and with a nonbreaking UTF-8 space (0xA0) in Opera.[*] Oh, and here's a really surprising bit: In Firefox, the whitespace at ❷ can also be replaced with a single, regular slash—yet the one at ❹ can't.

Moving on, the location marked ❸ is also of note. In this spot, NUL characters are ignored by most parsers, as are many types of whitespaces. Not long ago, WebKit browsers accepted a slash in this location, but recent parser improvements have eliminated this quirk.

Quote characters are a yet another topic of interest. Website developers know that single and double quotes can be used to put a string containing whitespaces or angle brackets in an HTML parameter, but it usually comes as a surprise that Internet Explorer also honors backticks (`) instead of real quotes in the location marked ❺. Similarly, few people realize that in any browser, an implicit whitespace is inserted after a quoted parameter, and that the explicit whitespace at ❻ can therefore be skipped without changing the meaning of the tag.

The security impact of these patterns is not always easy to appreciate, but consider an HTML filter tasked with scrubbing an *<img>* tag with an attacker-controlled *title* parameter. Let's say that in the input markup, this parameter is not quoted if it contains no whitespaces and angle brackets—a design that can be seen on a popular blogging site. This practice may appear safe at first, but in the following two cases, a malicious, injected *onerror* parameter will materialize inside a tag:

```
<img ... title=""onerror="alert(1)">
```

and

```
<img ... title=``onerror=`alert(1)`>
```

Yet another wonderful quote-related quirk in Internet Explorer makes this job even more complicated. While most browsers recognize quoting only when it is used at the beginning of a parameter value, Internet Explorer simply checks for any occurrence of an equal sign (=) followed by a quote and will parse this syntax in a rather unexpected way:

```
<img src=test.jpg?value=">Yes, we are still inside a tag!">
```

Interactions Between Multiple Tags

Parsing a single tag can be a daunting task, but as you might imagine, anomalous arrangements of multiple HTML tags will be even less predictable. Consider the following trivial example:

```
<i <b>
```

[*] The behavior exhibited by Opera is particularly sneaky: The Unicode whitespace is not recognized by many standard library functions used in server-side HTML sanitizers, such as *isspace(...)* in libc. This increases the risk of implementation glitches.

When presented with such syntax, most browsers only interpret *<i>* and treat the "<b" string as an invalid tag parameter. Firefox versions before 4, however, would automatically close the *<i>* tag first when encountering an angle bracket and, in the end, will interpret both *<i>* and *<b>*. In the spirit of fault tolerance, until recently WebKit followed that model, too.

A similar behavior can be observed in previous versions of Firefox when dealing with tag names that contain invalid characters (in this case, the equal sign). Instead of doing its best to ignore the entire block, the parser would simply reset and interpret the quoted tag:

```
<i="<b>">
```

The handling of tags that are not closed before the end of the file is equally fascinating. For example, the following snippet will prompt most browsers to interpret the *<i>* tag or ignore the entire string, but Internet Explorer and Opera use a different backtracking approach and will see *<b>* instead:

```
<i foo="<b>" [EOF]
```

In fact, Firefox versions prior to version 4 engaged in far-fetched reparsing whenever particular special tags, such as *<title>*, were not closed before the end of the document:

```
<title>This text will be interpreted as a title
<i>This text will be shown as document body!
[EOF]
```

The last two parsing quirks have interesting security consequences in any scenario where the attacker may be able to interrupt page load prematurely. Even if the markup is otherwise fairly well sanitized, the meaning of the document may change in a very unexpected way.

Explicit and Implicit Conditionals

To further complicate the job of HTML parsing, some browsers exhibit behaviors that can be used to conditionally skip some of the markup in a document. For example, in an attempt to help novice users of Microsoft's Active Server Pages development platform, Internet Explorer treats <% … %> blocks as a completely nonstandard comment, hiding any markup between these two character sequences. Another Internet Explorer–specific feature is explicit conditional expressions interpreted by the parser and smuggled inside standard HTML comment blocks:

```
<!--[if IE 6]>
  Markup that will be parsed only for Internet Explorer 6
<![endif]-->
```

Many other quirks of this type are related to the idiosyncrasies of SGML and XML. For example, due to the comment-handling behavior mentioned earlier in an aside, browsers disagree on how to parse !- and ?-directives (such as <*!DOCTYPE*> or <*?xml*>), whether to allow XML-style CDATA blocks in non-XHTML modes, and on what precedence to give to overlapping special parsing mode tags (such as "<*style*><*!–* </*style*> –>").

HTML Parsing Survival Tips

The set of parsing behaviors discussed in the previous sections is by no means exhaustive. In fact, an entire book has been written on this topic: Inquisitive readers are advised to grab *Web Application Obfuscation* (Syngress, 2011) by Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Heyes, and David Lindsay—and then weep about the fate of humanity. The bottom line is that building HTML filters that try to block known dangerous patterns, and allow the remaining markup as is, is simply not feasible.

The only reasonable approach to tag sanitization is to employ a realistic parser to translate the input document into a hierarchical in-memory document tree, and then scrub this representation for all unrecognized tags and parameters, as well as any undesirable tag/parameter/value configurations. At that point, the tree can be carefully reserialized into a well-formed, well-escaped HTML that will not flex any of the error correction muscles in the browser itself. Many developers think that a simpler design should be possible, but eventually they discover the reality the hard way.

Entity Encoding

Let's talk about character encoding again. As noted on the first pages of this chapter, certain reserved characters are generally unsafe inside text nodes and tag parameter values, and they will often lead to outright syntax errors in XHTML. In order to allow such characters to be used safely (and to allow a convenient way to embed high-bit text), a simple ampersand-prefixed, semicolon-terminated encoding scheme, known as entity encoding, is available to developers.

The most familiar use of this encoding method is the inclusion of certain predefined, named entities. Only a handful of these are specified for XML, but several hundred more are scattered in HTML specifications and supported by all modern browsers. In this approach, *<lt;* is used to insert a left angle bracket; *>* substitutes a right angle bracket; *&* replaces the ampersand itself; while, say, *→* is a nice Unicode arrow.

NOTE *In XHTML documents, additional named entities can be defined using the* <!ENTITY> *directive and made to resolve to internally defined strings or to the contents of an external file URL. (This last option is obviously unsafe if allowed when processing untrusted content; the resulting attack is sometimes called* External XML Entity*, or XXE for short.)*

In addition to the named entities, it is also possible to insert an arbitrary ASCII or Unicode character using a decimal *&#number;* notation. In this case, *<* maps to a left angle bracket; *>* substitutes a right one; and *😹* is, I kid you not, a Unicode 6.0 character named "smiling cat face with tears of joy." Hexadecimal notation can also be used if the number is prefixed with "x". In this variant, the left angle bracket becomes *<*, etc.

The HTML parser recognizes entity encoding inside text nodes and parameter values and decodes it transparently when building an in-memory representation of the document tree. Therefore, the following two cases are functionally identical:

```
<img src="http://www.example.com">
```

and

```
<img src="ht&#x74;p&#x3a;//www.example.com">
```

The following two examples, on the other hand, will not work as expected, as the encoding interferes with the structure of the tag itself:

```
<img src&#x3d;"http://www.example.com">
```

and

```
<img s&#x72;c="http://www.example.com">
```

The largely transparent behavior of entity encoding makes it important to correctly resolve it prior to making any security decisions about the contents of a document and, if applicable, to properly restore it in the sanitized output later on. To illustrate, the following syntax must be recognized as an absolute reference to a *javascript:* pseudo-URL and not to a cryptic fragment ID inside a relative resource named "./javascript&":

```
<a href="javascript&#x3a;alert(1)">
```

Unfortunately, even the simple task of recognizing and parsing HTML entities can be tricky. In traditional parsing, for example, entities may often be accepted even if the trailing semicolon is omitted, as long as the next character is not an alphanumeric. (In Firefox, dashes and periods are also accepted in entity names.) Numeric entities are even more problematic, as they may have an overlong notation with an arbitrary number of trailing zeros. Moreover, if the numerical value is higher than 2^{32}, the standard size of an integer on many computer architectures, the corresponding character may be computed incorrectly.

Developers working with XHTML should be aware of a potential pitfall in that dialect, too. Although HTML entities are not recognized in most of the special parsing modes, XHTML differs from traditional HTML in that tags such as *<script>* and *<style>* do not automatically toggle a special parsing mode on their own. Instead, an explicit *<![CDATA[…]]>* block around any scripts or stylesheets is required to achieve a comparable effect. Therefore, the following snippet with an attacker-controlled string (otherwise scrubbed for angle brackets, quotes, backslashes, and newlines) is perfectly safe in HTML, but not in XHTML:

```
<script>
  var tmp = 'I am harmless! &#x27;+alert(1);// Or am I?';
  ...
</script>
```

HTTP/HTML Integration Semantics

From Chapter 3, we recall that HTTP headers may give new meaning to the entire response (*Location*, *Transfer-Encoding*, and so on), change the way the payload is presented (*Content-Type*, *Content-Disposition*), or affect the client-side environment in other, auxiliary ways (*Refresh*, *Set-Cookie*, *Cache-Control*, *Expires*, etc.).

But what if an HTML document is delivered through a non-HTTP protocol or loaded from a local file? Clearly, in this case, there is no simple way to express or preserve this information. We can part with some of it easily, but parameters such as the MIME type or the character set are essential, and losing them forces browsers to improvise later on. (Consider, for example, that charsets such as UTF-7, UTF-16, and UTF-32 are not ASCII-compatible and, therefore, HTML documents can't even be parsed without determining which of these transformations needs to be used.)

The security consequences of the browser-level heuristics used to detect character sets and document types will be explored in detail in Chapter 13. Meanwhile, the problem of preserving protocol-level information within a document is somewhat awkwardly addressed by a special HTML directive, *<meta http-equiv=…>*. By the time the browser examines the markup, many content-handling decisions must have already been made, but some tweaks are still on the table; for example, it may be possible to adjust the charset to a generally compatible value or to specify *Refresh*, *Set-Cookie*, and caching directives.

As an illustration of permissible syntax, consider the following directive that, when appearing in an 8-bit ASCII document, will clarify for the browser that the charset of the document is UTF-8 and not, say, ISO-8859-1:

```
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
```

On the flip side, all of the following directives will fail, because at this point it is too late to switch to an incompatible UTF-32 encoding, change the document type to a video format, or execute a redirect instead of parsing the file:

```
<meta http-equiv="Content-Type" content="text/html;charset=utf-32">
<meta http-equiv="Content-Type" content="video/mpeg">
<meta http-equiv="Location" content="http://www.example.com">
```

Be mindful that when *http-equiv* values conflict with each other, or contradict the HTTP headers received from the server earlier on, their behavior is not consistent and should not be relied upon. For example, the first supported *charset=* value usually prevails (and HTTP headers have precedence over *<meta>* in this case), but with several conflicting *Refresh* values, the behavior is highly browser-specific.

NOTE *Some browsers will attempt to speculatively extract <meta http-equiv> information before actually parsing the document, which may lead to embarrassing mistakes. For example, a security bug recently fixed in Firefox 4 caused the browser to interpret the following statement as a character set declaration:* <meta http-equiv="Refresh" content="10;http://www.example.com/charset=utf-7">.[6]

Hyperlinking and Content Inclusion

One of the most important and security-relevant features of HTML is, predictably, the ability to link to and embed external content. HTTP-level features such as *Location* and *Refresh* aside, this can be accomplished in a couple of straightforward ways.

Plain Links

The following markup demonstrates the most familiar and most basic method for referencing external content from within a document:

```
<a href="http://www.example.com/">Click me!</a>
```

This hyperlink may point to any of the browser-recognized schemes, including pseudo-URLs (*data:*, *javascript:*, and so on) and protocols handled by external applications (such as *mailto:*). Clicking on the text (or any HTML elements) nested inside such a *<a href=...>* block will typically prompt the browser to navigate away from the linking document and go to the specified location, if meaningfully possible for the protocol used.

An optional *target* parameter may be used to target other windows or document views for navigation. The parameter must specify the name of the target view. If the name cannot be found, or if access is denied, the default behavior is typically to open a new window instead. The conditions in which access may be denied are the topic of Chapter 11.

Four special target names can be used, too (as shown on the left of Figure 4-1): *_blank* always opens a brand-new window, *_parent* navigates a higher-level view that embeds the link-bearing document (if any), and *_top* always navigates the top-level browser window, no matter how many document embedding levels are in between. Oh, right, the fourth special target, *_self*, is identical to not specifying a value at all and exists for no reason whatsoever.

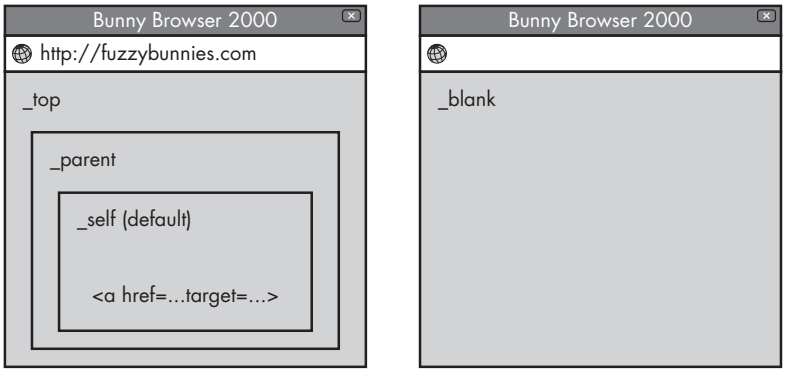


Figure 4-1: Predefined targets for hyperlinks

Forms and Form-Triggered Requests

An HTML form can be thought of as an information-gathering hyperlink: When the "submit" button is clicked, a dynamic request is constructed on the fly from the data collected via any number of input fields. Forms allow user input and files to be uploaded to the server, but in almost every other way, the result of submitting a form is similar to following a normal link.

A simple form markup may look like this:

```
<form method=GET action="/process_form.cgi">
  Given name: <input type=text name=given>
  Family name: <input type=text name=family>
  ...
  <input type=submit value="Click here when done!">
</form>
```

The *action* parameter works like the *href* value used for normal links, with one minor difference: If the value is absent, the form will be submitted to the location of the current document, whereas any destination-free *<a>* links will simply not work at all. An optional *target* parameter may also be specified and will behave as outlined in the previous section.

NOTE *Unusually, unlike* <a> *tags, forms cannot be nested inside each other, and only the top-level* <form> *tag will remain operational in such a case.*

When the *method* value is set to GET or is simply not present at all, all the nested field names and their current values will be escaped using the familiar percent-encoding scheme outlined in Chapter 2, but with two rather arbitrary differences. First, the space character (0x20) will be substituted with the plus

sign, rather than encoded as "%20". Second, following from this, any existing plus signs need to be encoded as "%2B", or else they will be misinterpreted as spaces.

Encoded *name=value* pairs are then delimited with ampersands and combined into a single string, such as this:

```
given=Erwin+Rudolf+Josef+Alexander&family=Schr%C3%B6dinger
```

The resulting value is inserted into the query part of the destination URL (replacing any existing contents of that section) and submitted to the server. The received response is then shown to the user in the targeted viewport.

The situation is a bit more complicated if the *method* parameter is set to POST. For that type of HTTP request, three data submission formats are available. In the default mode (referred to as *application/x-www-form-urlencoded*), the message is constructed the same way as for GET but is transmitted in the request payload instead, leaving the query string and all other parts of the destination URL intact.*

The existence of the second POST submission mode, triggered by specifying *enctype="text/plain"* on the *<form>* tag, is difficult to justify. In this mode, field names and values will not be percent encoded at all (but, depending on the browser, plus signs may be used to substitute for spaces), and a newline delimiter will be used in place of an ampersand. The resulting format is essentially useless, as it can't be parsed unambiguously: Form-originating newlines and equal signs are indistinguishable from browser inserted ones.

The last mode is triggered with *enctype="multipart/form-data"* and must be used whenever submitting user-selected files through a form (which is possible with a special *<input type="file">* tag). The resulting request body consists of a series of short MIME messages corresponding to every submitted field.† These messages are delimited with a client-selected random, unique boundary token that should otherwise not appear in the encapsulated data:

```
POST /process_form.cgi HTTP/1.1
…
Content-Type: multipart/form-data; boundary=random1234

--random1234
Content-Disposition: form-data; name="given"

Erwin Rudolf Josef Alexander
--random1234
Content-Disposition: form-data; name="family"
```

* This has the potential for confusion, as the same parameter may appear both in the query string and in the POST payload. There is no consistency in how various server-side web applications frameworks resolve this conflict.

† MIME (Multipurpose Internet Mail Extensions) is a data format intended for encapsulating and safely transmitting various types of documents in email messages. The format makes several unexpected appearances in the browser world. For example, *Content-Type* file format identifiers also have unambiguous MIME roots.

```
Schrödinger
--random1234
Content-Disposition: form-data; name="file"; filename="cat_names.txt"
Content-Type: text/plain

(File contents follow)
--random1234--
```

Despite the seemingly open-ended syntax of the tag, other request methods and submission formats are not supported by any browser, and this is unlikely to change. For a short while, the HTML5 standard tried to introduce PUT and DELETE methods in forms, but this proposal was quickly shot down.

Frames

Frames are a form of markup that allows the contents of one HTML document to be displayed in a rectangular region of another, embedding page. Several framing tags are supported by modern browsers, but the most common way of achieving this goal is with a hassle-free and flexible inline frame:

```
<iframe src="http://www.example.com/"></iframe>
```

In traditional HTML documents, this tag puts the parser in one of the special parsing modes, and all text between the opening and the closing tag will simply be ignored in frame-aware browsers. In legacy browsers that do not understand *<iframe>*, the markup between the opening and closing tags is processed normally, however, offering a decidedly low-budget, conditional rendering directive. This conditional behavior is commonly used to provide insightful advice such as "This page must be viewed in a browser that supports frames."

The frame is a completely separate document view that in many aspects is identical to a new browser window. (It even enjoys its own JavaScript execution context.) Like browser windows, frames can be equipped with a *name* parameter and then targeted from *<a>* and *<form>* tags.

The constraints on the *src* URL for framed content are roughly similar to the rules enforced on regular links. This includes the ability to point frames to javascript: or to load externally handled protocols that leave the frame empty and open the target application in a new process.

Frames are of special interest to web security, as they allow almost unconstrained types of content originating from unrelated websites to be combined onto a single page. We will have a second look at the problems associated with this behavior in Chapter 11.

Type-Specific Content Inclusion

In addition to content-agnostic link navigation and document framing, HTML also provides multiple ways for a more lightweight inclusion of several predefined types of external content.

Images

Image files can be retrieved and displayed on a page using *<img>* tags, via stylesheets, and through a legacy *background=* parameter on markup such as *<body>* or *<table>*.

The most popular image type on the Internet is a lossy but very efficient JPEG file, followed by lossless and more featured (but slower) PNG. An increasingly obsolete lossless GIF format is also supported by every browser, and so is the rarely encountered and usually uncompressed Windows bitmap file (BMP). An increasing number of rendering engines support SVG, an XML-based vector graphics and animation format, too, but the inclusion of such images through the *<img>* tag is subject to additional restrictions.

The list of recognized image types can be wrapped up with odds and ends such as Windows metafiles (WMF and EMF), Windows Media Photo (WDP and HDP), Windows icons (ICO), animated PNG (APNG), TIFF images, and—more recently—WebP. Browser support for these is far from universal, however.

Cascading stylesheets

These text-based files can be loaded with a *<link rel=stylesheet href=…>* tag—even though *<style src=…>* would be a more intuitive choice—and may redefine the visual aspects of almost any other HTML tag within their parent document (and in some cases, even include embedded JavaScript). The syntax and function of CSS are the subject of Chapter 5.

In the absence of the appropriate *charset* value in the *Content-Type* header for the downloaded stylesheet, the encoding according to which this subresource will be interpreted can be specified by the including party through the *charset* parameter of the *<link>* tag.

Scripts

Scripts are text-based programs included with *<script>* tags and are executed in a manner that gives them full control over the host document. The primary scripting language for the Web is JavaScript, although an embedded version of Visual Basic is also supported in Internet Explorer and can be used at will. Chapter 6 takes an in-depth look at client-side scripts and their capabilities.

As with CSS, in the absence of valid *Content-Type* data, the charset according to which the script is interpreted may be controlled by the including party.

Plug-in content

This category spans miscellaneous binary files included with *<embed>* or *<object>* tags or via an obsolete, Java-specific *<applet>* tag. Browser plug-in content follows its own security rules, which are explored to some extent in Chapters 8 and 9. In many cases, it is safe to consider plug-in-supported content as equivalent to or more powerful than JavaScript.

NOTE *The standard permits certain types of browser-supported documents, such as* text/html *or* text/plain*, to be loaded through* <object> *tags, in which case they form a close equivalent of* <iframe>*. This functionality is not used in practice, and the rationale behind it is difficult to grasp.*

Other supplementary content
: This category includes various rendering cues that may or may not be honored by the browser; they are most commonly provided through *<link>* directives. Examples include website icons (known as "favicons"), alternative versions of a page, and chapter navigation links.

Several other once-supported content inclusion methods, such as the *<bgsound>* tag for background music, were commonplace in the past but have fallen out of grace. On the other hand, as a part of HTML5, new tags such as *<video>* and *<audio>* are expected to gain popularity soon.

There is relatively little consistency in what URL schemes are accepted for type-specific content retrieval. It should be expected that protocols routed to external applications will be rejected, as they do not have a sensible meaning in this context, but beyond this, not many assumptions should be made. As a security precaution, most browsers will also reject scripting-related schemes when loading images and stylesheets, although Internet Explorer 6 and Opera do not follow this practice. As of this writing, *javascript:* URLs are also permitted on *<embed>* and *<applet>* tags in Firefox but not, for example, on *<img>*.

For almost all of the type-specific content inclusion methods, *Content-Type* and *Content-Disposition* headers provided by the server will typically be ignored (perhaps except for the *charset=* value), as may be the HTTP response code itself. It is best to assume that whenever the body of any server-provided resource is even vaguely recognizable as one of the data formats enumerated in this section, it may be interpreted as such.

A Note on Cross-Site Request Forgery

On all types of cross-domain navigation, the browser will transparently include any ambient credentials; consequently, to the server, a request legitimately originating from its own client-side code will appear roughly the same as a request originating from a rogue third-party site, and it may be granted the same privileges.

Applications that fail to account for this possibility when processing any sensitive, state-changing requests are said to be vulnerable to *cross-site request forgery* (*XSRF* or *CSRF*). This vulnerability can be mitigated in a number of ways, the most common of which is to include a secret user- and session-specific value on such requests (as an additional query parameter or a hidden form field). The attacker will not be able to obtain this value, as read access to cross-domain documents is restricted by the same-origin policy (see Chapter 9).

Security Engineering Cheat Sheet

Good Engineering Hygiene for All HTML Documents

☑ Always output consistent, valid, and browser-supported *Content-Type* and *charset* information to prevent the document from being interpreted contrary to your original intent.

When Generating HTML Documents with Attacker-Controlled Bits

This task is difficult to perform consistently across the entire web application, and it is one of the most significant sources of web application security flaws. Consider using context-sensitive auto-escaping frameworks, such as *JSilver* or *CTemplate*, to automate it. If that is not possible, read on.

☑ **User-supplied content in text body:** Always entity-encode "<", ">", and "&". Note that certain other patterns may be dangerous in certain non-ASCII-compatible output encodings. If applicable, consult Chapter 13.

Keep in mind that some Unicode metacharacters (e.g., U+202E) alter the direction or flow of the subsequent text. It may be desirable to remove them in particularly sensitive uses.

☑ **Tag-specific *style* and *on** parameters:** Multiple levels of escaping are required. This practice is extremely error prone, meaning not really something to attempt. If it is absolutely unavoidable, review the cheat sheets in Chapters 5 and 6.

☑ **All other HTML parameter values:** Always use quotes around attacker-controlled input. Entity-encode "<", ">", "&", and any stray quotes. Remember that some parameters require additional validation. For URLs, see the cheat sheet in Chapter 2.

Never attempt to blacklist known bad values in URLs or any other parameters; doing so will backfire and may lead to script execution flaws.

☑ **Special parsing modes (e.g., <*script*> and <*style*> blocks):** For values appearing inside quoted strings, replace quote characters, backslash, "<", ">", and all nonprintable characters with language-appropriate escape codes. For values appearing outside strings, exercise extreme caution and allow only carefully validated, known, alphanumeric values.

In XHTML mode, remember to wrap the entire script section in a CDATA block. Avoid cases that require multiple levels of encoding, such as building parameters to the JavaScript *eval(...)* function using attacker-supplied strings. Never place user-controlled data inside HTML comments, !-type or ?-type tags, and other nonessential or unusually parsed blocks.

When Converting HTML to Plaintext

☑ A common mistake is to strip only well-formed tags. Remember that all left-angle brackets must be removed, even if no matching right-angle bracket is found. To minimize the risk of errors, always entity-escape angle brackets and ampersands in the generated output, too.

When Writing a Markup Filter for User Content

- ☑ Read this chapter carefully. Use a reasonably robust HTML parser to build an in-memory document tree. Walk the tree, removing any unrecognized or unnecessary tags and parameters and scrubbing any undesirable tags/parameters/value combinations.

  When done, reserialize the document, making sure to apply proper escaping rules to parameter values and text content. (See the first tip on this cheat sheet.) Be aware of the impact of special parsing modes.
- ☑ Because of the somewhat counterintuitive namespace interactions with JavaScript, do not allow *name* and *id* parameters on user-supplied markup—at least not without reading Chapter 6 first.
- ☑ Do not attempt to sanitize an existing, serialized document in place. Doing so inevitably leads to security problems.

5

CASCADING STYLE SHEETS

As the Web matured through the 1990s, website developers increasingly needed a consistent and flexible way to control the appearance of HTML documents; the collection of random, vendor-specific tag parameters available at the time simply would not do. After reviewing several competing proposals, W3C eventually settled on *Cascading Style Sheets (CSS)*, a fairly simple text-based page appearance description language proposed by Håkon Wium Lie.

The initial CSS level 1 specification saw the light of day by the end of 1996,[1] but further revisions of this document continued until 2008. The initial draft of CSS level 2 followed in December 1998 and has yet to be finalized as of 2011. The work on the most recent iteration, level 3, started in 2005 and also continues to this day. Although most of the individual features envisioned for CSS2 and CSS3 have been adopted by all modern browsers after years of trial and error, many subtle details vary significantly from one implementation to another, and the absence of a finalized standard likely contributes to this.

Despite the differences from one browser to another, CSS is a very powerful tool. With only a couple of constraints, stylesheets permit almost every HTML tag to be scaled, positioned, and decorated nearly arbitrarily, thereby overcoming the constraints originally placed on it by the underlying markup language; in some implementations, JavaScript programs can be embedded in the CSS presentation directives as well. The job of placing user-controlled values inside stylesheets, or recoding any externally provided CSS, is therefore of great interest to web application security.

Basic CSS Syntax

Stylesheets can be placed in an HTML document in three ways: inlined globally for the entire document with a *<style>* block, retrieved from an external URL via the *<link rel=stylesheet>* directive, or attached to a specific tag using the *style* parameter. In addition, XML-based documents (including XHTML) may also leverage a little-known *<?xml-stylesheet href=... ?>* directive to achieve the same goal.

The first two methods of inclusion require a fully qualified stylesheet consisting of any number of selectors (directives describing which HTML tags the following ruleset will apply to) followed by semicolon-delimited *name: value* rules between curly brackets. Here is a simple example of such syntax, defining the appearance of *<img>*, **, and * * tags:

```
img {
  border-size: 1px;
  border-style: solid;
}

span, div {
  color: red;
}
```

Selectors can reference a particular type of a tag (such as *img*), a period-prefixed name of a class of tags (for example, *.photos*, which will apply to all tags with an inline *class=photos* parameter), or a combination of both (*img.company_logo*). Selector suffixes such as *:hover* or *:visited* may also be used to make the selector match only under certain circumstances, such as when the mouse hovers over the content or when a particular displayed hyperlink has already been visited before.

So-called *complex selectors*[2] are an interesting feature introduced in CSS2 and extended in CSS3. They allow any given ruleset to apply only to tags with particular strings appearing in parameter values or that are positioned in a particular relation to other markup. One example of such a selector is this:

```
a[href^="ftp:"] {
  /* Styling applicable only to FTP links. */
}
```

NOTE *Oh, while we are at it: As evident in this example, C-style /*...*/ comment blocks are permitted in CSS syntax anywhere outside a quoted string. On the flip side, //-style comments are not recognized at all.*

Property Definitions

Inside the *{ … }* block that follows a selector, as well as inside the *style* parameter attached to a specific tag, any number of *name: value* rules can be used to redefine almost every aspect of how the affected markup is displayed. Visibility, shape, color, screen position, rendering order, local or remote typeface, and even any additional text (*content* property supported on certain pseudo-classes) and mouse cursor shape are all up for grabs.[*] Simple types of automation, such as counters for numbered lists, are available through CSS rules as well.

Property values can be formatted as the following:

- **Raw text** This method is used chiefly to specify numerical values (with optional units), RGB vectors and named colors, and other predefined keywords ("absolute," "left," "center," etc.).
- **Quoted strings** Single or double quotes should be placed around any nonkeyword values, but there is little consistency in how this rule is enforced. For example, quoting is not required around typeface names or certain uses of URLs, but it is necessary for the aforementioned *content* property.
- **Functional notation** Two parameter-related pseudo-functions are mentioned in the original CSS specification: *rgb(…)*, for converting individual RGB color values into a single color code, and *url(…)*, required for URLs in most but not all contexts. On top of this, several more pseudo-functions have been rolled out in recent years, including *scale(…)*, *rotate(…)*, or *skew(…)*.

  A proprietary *expression(…)* function is also available in Internet Explorer; it permits JavaScript statements to be inserted within CSS. This function is one of the most important reasons why attacker-controlled stylesheets can be a grave security risk.

@ Directives and XBL Bindings

In addition to selectors and properties, several @-prefixed directives are recognized in stand-alone stylesheets. All of them modify the meaning of the stylesheet; for example, by specifying the namespace or the display media that the stylesheet should be applied to. But two special directives also affect the behavior of the parsing process. The first of these is *@charset*, which sets the charset of the current CSS block; the other is *@import*, which inserts an external file into the stylesheet.

[*] The ability to redefine mouse cursors using an arbitrary bitmap has predictably resulted in some security bugs. An oversized cursor combined with script-based mouse position tracking could be used to obscure or replace important elements of the browser UI and trick the user into doing something dangerous.

The *@import* directive itself serves as a good example of the idiosyncrasies of CSS parsing; the parser views all of the following examples as equivalent:

```
@import "foo.css";
@import url('foo.css');
@import'foo.css';
```

In Firefox, external content directives, including JavaScript code, may be also loaded from an external source using the *-moz-binding* property, a vendor-specific way to weave XML Binding Language[3] files (an obscure method of providing automation to XML content) into the document. There is some talk of supporting XBL in other browsers, too, at which point the name of the property would change and the XSS risk may or may not be addressed in some way.

NOTE *As can be expected, the handling of pseudo-URLs in* @import, url(...) *and other CSS-based content inclusion schemes is a potential security risk. While most current browsers do not accept scripting-related schemes in these contexts, Internet Explorer 6 allows them without reservations, thereby creating a code injection vector if the URL is not validated carefully enough.*

Interactions with HTML

It follows from the discussion in the previous chapter that for any stylesheets inlined in HTML documents, HTML parsing is performed first and is completely independent of CSS syntax rules. Therefore, it is unsafe to place certain HTML syntax characters inside CSS properties, as in the following example, even when quoted properly. A common mistake is permitting this:

```
<style>
some_descriptor {
  background: url('http://www.example.com/</style><h1>Gotcha!');
}
</style>
```

We'll discuss a way to encode problematic characters in stylesheets shortly, but first, let's have a quick look at another very distinctive property of CSS.

Parser Resynchronization Risks

An undoubtedly HTML-inspired behavior that sets CSS apart from most other languages is that compliant parsers are expected to continue after encountering a syntax error and restart at the next matching curly bracket (some superficial nesting-level tracking is mandated by the spec). In particular, the following stylesheet snippet, despite being obviously malformed, will still apply the specified border style to all <*img*> tags:

```
a {
  $$$ This syntax makes absolutely no sense $$$
  !(@*#)!!@ 123
}
```

```
img {
  border: 1px solid red;
}
```

This unusual behavior creates an opportunity to exploit parser incompatibilities in an interesting way: If there is any way to derail a particular CSS implementation with inputs that seem valid to other parsers, the resynchronization logic may cause the attacked browser to resume parsing at an incorrect location, such as in the middle of an attacker-supplied string.

A naïve illustration of this issue may be Internet Explorer's support for multiline string literals. In this browser, it is seemingly safe not to scrub CR and LF characters in user-supplied CSS strings, so some webmasters may allow it. Unfortunately, the same pattern will cause any other browser to resume at an unexpected offset and interpret the *evil_rule* ruleset:

```
some_benign_selector {
  content: 'Attacker-controlled text...
            } evil_rule { margin-left: -1000px; }';
}
```

The support for multiline strings is a Microsoft-specific extension, and the aforementioned problem is easily fixed by avoiding such noncompliant syntax to begin with. Unfortunately, other desynchronization risks are introduced by the standard itself. For example, recall complex selectors: This CSS3 syntax makes no sense to pre-CSS3 parsers. In the following example, an older implementation may bail out after encountering an unexpected angle bracket and resume parsing from the attacker-supplied *evil_rule* instead:

```
a[href^='} evil_rule { margin-left: -1000px; }'] {
  /* Harmless, validated rules here. */
}
```

The still-popular browser Internet Explorer 6 would be vulnerable to this trick.

Character Encoding

To make it possible to quote reserved or otherwise problematic characters inside strings, CSS offers an unorthodox escaping scheme: a backslash (\) followed by one to six hexadecimal digits. For example, according to this scheme, the letter *e* may be encoded as "\65", "\065", or "\000065". Alas, only the last syntax, "\000065", will be unambiguous if the next character happens to be a valid hexadecimal digit; encoding "teak" as "t\65ak" would not work as expected, because the escape sequence would be interpreted as "\65A", an Arabic sign in the Unicode character map.

To avoid this problem, the specification embraces an awkward compromise: A whitespace can follow an escape sequence and will be interpreted as a terminator, and then removed from the string (e.g., "t\65 ak"). Regrettably, more familiar and predictable fixed-length C-style escape sequences such as *\x65* cannot be used instead.

In addition to the numerical escaping scheme, it is also possible to place a backslash in front of a character that is not a valid hexadecimal digit. In this case, the subsequent character will be treated as a literal. This mechanism is useful for encoding quote characters and the backslash itself, but it should not be used to escape HTML control characters such as angle brackets. The aforementioned precedence of HTML parsing over CSS parsing renders this approach inadequate.

In a bizarre twist, due to somewhat ambiguous guidance in the W3C drafts, many CSS parsers recognize arbitrary escape sequences in locations other than quote-enclosed strings. To add insult to injury, in Internet Explorer, the substitution of these sequences apparently takes place before the pseudo-function syntax is parsed, effectively making the following two examples equivalent:

```
color: expression(alert(1))
```

```
color: expression\028 alert \028 1 \029 \029
```

Even more confusingly, in a misguided bid to maintain fault tolerance, Microsoft's implementation does not recognize backslash escape codes inside *url(...)* values; this is, once more, to avoid hurting the feelings of users who type the wrong type of a slash when specifying a URL.

These and similar quirks make the detection of known dangerous CSS syntax extremely error prone.

Security Engineering Cheat Sheet

When Loading Remote Stylesheets

- ☑ You are linking the security of your site to the originating domain of the stylesheet. Even in browsers that do not support JavaScript expressions inside stylesheets, features such as conditional selectors and *url(...)* references can be used to exfiltrate portions of your site.[4]
- ☑ When in doubt, make a local copy of the data instead.
- ☑ On HTTPS sites, require stylesheets to be served over HTTPS as well.

When Putting Attacker-Controlled Values into CSS

- ☑ **Strings and URLs inside stand-alone blocks.** Always use quotes. Backslash-escape all control characters (0x00–0x1F), "\", "<", ">", "{", "}", and quotes using numerical codes. It is also preferable to escape high-bit characters. For URLs, consult the cheat sheet in Chapter 2 to avoid code injection vulnerabilities.
- ☑ **Strings in *style* parameters**. Multiple levels of escaping are involved. The process is error prone, so do not attempt it unless absolutely necessary. If it is unavoidable, apply the above CSS escaping rules first and then apply HTML parameter encoding to the resulting string.
- ☑ **Nonstring attributes**. Allow only whitelisted alphanumeric keywords and carefully validated numerical values. Do not attempt to reject known bad patterns instead.

When Filtering User-Supplied CSS

- ☑ Remove all content outside of functional rulesets. Do not preserve or generate user-controlled comment blocks, @-directives, and so on.
- ☑ Carefully validate selector syntax, permitting only alphanumerics; underscores; whitespaces; and correctly positioned colons, periods, and commas before "{". Do not permit complex text-matching selectors; they are unsafe.
- ☑ Parse and validate every rule in the { … } block. Permit only whitelisted properties with well-understood consequences and confirm that they take expected, known safe values. Note that strings passed to certain properties may sometimes be interpreted as URLs even in the absence of a *url(...)* wrapper.
- ☑ Encode every parameter value using the rules outlined earlier in this section. Bail out on any syntax abnormalities.
- ☑ Keep in mind that unless specifically prevented from doing so, CSS may position user content outside the intended drawing area or redefine the appearance of any part of the UI of your application. The safest way to avoid this problem is to display the untrusted content inside a separate frame.

When Allowing User-Specified Class Values on HTML Markup

- ☑ Ensure that user-supplied content can't reuse class names that are used for any part of the application UI. If a separate frame is not being used, it's advisable to maintain separate namespace prefixes.

6

BROWSER-SIDE SCRIPTS

The first browser scripting engine debuted in Netscape Navigator around 1995, thanks to the work of Brendan Eich. The integrated Mocha language, as it was originally called, gave web developers the ability to manipulate HTML documents, display simple, system-level dialogs, open and reposition browser windows, and use other basic types of client-side automation in a hassle-free way.

While iterating through beta releases, Netscape eventually renamed Mocha LiveScript, and after an awkward branding deal was struck with Sun Microsystems, JavaScript was chosen as the final name. The similarities between Brendan's Mocha and Sun's Java were few, but the Netscape Corporation bet that this odd marketing-driven marriage would secure JavaScript's dominance in the more lucrative server world. It made this sentiment clear

in a famously confusing 1995 press release that introduced the language to the world and immediately tried to tie it to an impressive range of random commercial products:[1]

> **Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet**
>
> [. . .]
>
> Netscape Navigator Gold 2.0 enables developers to create and edit JavaScript scripts, while Netscape LiveWire enables JavaScript programs to be installed, run and managed on Netscape servers, both within the enterprise and across the Internet. Netscape LiveWire Pro adds support for JavaScript connectivity to high-performance relational databases from Illustra, Informix, Microsoft, Oracle and Sybase. Java and JavaScript support are being built into all Netscape products to provide a unified, front-to-back, client/server/tool environment for building and deploying live online applications.

Despite Netscape's misplaced affection for Java, the value of JavaScript for client-side programming seemed clear, including to the competition. In 1996 Microsoft responded by shipping a near-verbatim copy of JavaScript in Internet Explorer 3.0 along with a counterproposal of its own: a Visual Basic–derived language dubbed VBScript. Perhaps because it was late to the party, and perhaps because of VBScript's clunkier syntax, Microsoft's alternative failed to gain prominence or even any cross-browser support. In the end, JavaScript secured its position in the market, and in part due to Microsoft's failure, no new scripting languages have been attempted in mainstream browsers since.

Encouraged by the popularity of the JavaScript language, Netscape handed over some of the responsibility for maintaining it to an independent body, the European Computer Manufacturers Association (ECMA). The new overseers successfully released ECMAScript, 3rd edition in 1999[2] but had substantially more difficulty moving forward from there. The 4th edition, an ambitious overhaul of the language, was eventually abandoned after several years of bickering between the vendors, and a scaled-down 5th edition,[3] published in 2009, still enjoys only limited (albeit steadily improving) browser support. The work on a new iteration, called "Harmony," begun in 2008, still has not been finalized. Absent an evolving and widely embraced standard, vendor-specific extensions of the language are common, but they usually cause only pain.

Basic Characteristics of JavaScript

JavaScript is a fairly simple language meant to be interpreted at runtime. It has vaguely C-influenced syntax (save for pointer arithmetic); a straightforward classless object model, said to be inspired by a little-known programming language named Self; automatic garbage collection; and weak, dynamic typing.

JavaScript as such has no built-in I/O mechanisms. In the browser, limited abilities to interact with the host environment are offered through a set

of predefined methods and properties that map to native code inside the browser, but unlike what can be seen in many other programming languages, these interfaces are fairly limited and purpose built.

Most of the core features of JavaScript are fairly unremarkable and should be familiar to developers already experience with C, C++, or, to a lesser extent, Java. A simple JavaScript program might look like this:

```
var text = "Hi mom!";

function display_string(str) {
  alert(str);
  return 0;
}

// This will display "Hi mom!".
display_str(text);
```

Because it is beyond the scope of this book to provide a more detailed overview of the semantics of JavaScript, we'll summarize only some of its more unique and security-relevant properties later in this chapter. For readers looking for a more systematic introduction to the language, Marijn Haverbeke's *Eloquent JavaScript* (No Starch Press, 2011) is a good choice.

Script Processing Model

Every HTML document displayed in a browser—be it in a separate window or in a frame—is given a separate instance of the JavaScript execution environment, complete with an individual namespace for all global variables and functions created by the loaded scripts. All scripts executing in the context of a particular document share this common sandbox and can also interact with other contexts through browser-supplied APIs. Such cross-document interactions must be done in a very explicit way; accidental interference is unlikely. Superficially, script-isolation rules are reminiscent of the process-compartmentalization model in modern multitasking operating systems but a lot less inclusive.

Within a particular execution context, all encountered JavaScript blocks are processed individually and almost always in a well-defined order. Each code block must consist of any number of self-contained, well-formed syntax units and will be processed in three distinct, consequent steps: parsing, function resolution, and code execution.

Parsing

The parsing stage validates the syntax of the script block and, usually, converts it to an intermediate binary representation, which can be subsequently executed at a more reasonable speed. The code has no global effects until this step completes successfully. In case of syntax errors, the entire problematic block is abandoned, and the parser proceeds to the next available chunk of code.

To illustrate the behavior of a compliant JavaScript parser, consider the following HTML snippet:

block #1:

```
<script>
var my_variable1 = 1;
var my_variable2 =
</script>
```

block #2:

```
<script>
2;
</script>
```

Contrary to what developers schooled in C may be accustomed to, the above sequence is not equivalent to the following snippet:

```
<script>
var my_variable1 = 1;
var my_variable2 = 2;
</script>
```

This is because *<script>* blocks are not concatenated before parsing. Instead, the first script segment will simply cause a syntax error (an assignment with a missing right-hand value), resulting in the entire block being ignored and not reaching execution stage. The fact that the whole segment is abandoned before it can have any global side effects also means that the original example is not equivalent to this:

```
<script>
var my_variable1 = 1;
</script>

<script>
2;
</script>
```

This sets JavaScript apart from many other scripting languages such as Bash, where the parsing stage is not separated from execution in such a strong way.

What will happen in the original example provided earlier in this section is that the first block will be ignored but the second one (*<script>2;</script>*) will be parsed properly. That second block will amount to a no-op when executed, however, because it uses a pure, numerical expression as a code statement.

Function Resolution

Once the parsing stage is completed successfully, the next step involves registering every named, global function that the parser found within the currently processed block. Past this point, each function found will be reachable

from the subsequently executed code. Because of this extra pre-execution step, the following syntax will work flawlessly (contrary to what programmers may be accustomed to in C or C++, *hello_world()* will be registered before the first code statement—a call to said function—is executed):

```
<script>
hello_world();

function hello_world() {
  alert('Hi mom!');
}
</script>
```

On the other hand, the modified example below will not have the desired effect:

```
<script>
hello_world();
</script>

<script>
function hello_world() {
  alert('Hi mom!');
}
</script>
```

This modified case will fail with a runtime error because individual blocks of code are not processed simultaneously but, rather, are looked at based on the order in which they are made available to the JavaScript engine. The block that defines *hello_world()* will not yet be parsed when the first block is already executing.

To further complicate the picture, the mildly awkward global name resolution model outlined here applies only to functions, not to variable declarations. Variables are registered sequentially at execution time, in a way similar to other interpreted scripting languages. Consequently, the following code sample, which merely replaces our global *hello_world()* with an unnamed function assigned to a global variable, will not work as planned:

```
<script>
hello_world();

var hello_world = function() {
  alert('Hi mom!');
}
</script>
```

In this case, the assignment to the *hello_world* variable will not be done by the time the *hello_world()* call is attempted.

Code Execution

Once function resolution is completed, the JavaScript engine normally proceeds with the ordered execution of all statements outside of function blocks. The execution of a script may fail at this point due to an unhandled exception or for a couple of other, more esoteric reasons. If such an error is encountered, however, any resolved functions within the offending code block will remain callable, and any effects of the already executed code will persist in the current scripting context.

Exception recovery and several other JavaScript execution characteristics are illustrated by the following lengthy but interesting code snippet:

```
<script>
function not_called() {
  return 42;
}

function hello_world() {
  alert("With this program, anything is possible!");
  do_stuff();
}

alert("Welcome to our demo application.");

hello_world();

alert("Thank you, come again.");
</script>

<script>
alert("Now that you are done, how about a nice game of chess?");
</script>
```

This function will not execute, because it's not called from anywhere.

This function will execute only when called. It will show a dialog, but then will throw an exception due to an unresolved reference to a function named `do_stuff()`.

The execution of the program will start from this statement.

The "With this..." message will be displayed next.

This code will not be reached due to an unhandled exception triggered inside `hello_world()`.

The previous exception will not prevent this independent block from executing next.

Try to follow this example on your own and see if you agree with the annotations provided on the right.

As should be evident from this exercise, any unexpected and unhandled exceptions have an unusual consequence: They may leave the application in an inconsistent but still potentially executable state. Because exceptions are meant to prevent error propagation caused by unanticipated errors, this design is odd—especially given that on many other fronts (such as the ban on *goto* statements), JavaScript exhibits a more fundamentalist stance.

Execution Ordering Control

In order to properly analyze the security properties of certain common web application design patterns, it is important to understand the JavaScript engine's execution ordering and timing model. Thankfully, this model is remarkably sane.

Virtually all JavaScript living within a particular execution context is executed synchronously. The code can't be reentered due to an external event while it is still executing, and there is no support for threads that would be able to simultaneously modify any shared memory. While the execution engine is busy, the processing of events, timers, page navigation requests, and so on, is postponed; in most cases, the entire browser, or at least the HTML renderer, will also remain largely unresponsive. Only once the execution stops and the scripting engine enters an idle state will the processing of queued events resume. At this point, the JavaScript code may be entered again.

Further, JavaScript offers no *sleep(...)* or *pause(...)* function to temporarily release the CPU and later resume execution from the same location. Instead, if a programmer desires to postpone the execution of a script, it is necessary to register a timer to initiate a new execution flow later on. This flow will need to start at the beginning of a specified handler function (or at the beginning of an ad hoc, self-contained snippet of code provided when setting up a timer). Although these design decisions can be annoying, they substantially reduce the risk of race conditions in the resulting code.

NOTE *There are several probably unintentional loopholes in this synchronous execution model. One of them is the possibility of code execution while the execution of another piece of JavaScript is temporarily suspended after calling* alert(...) *or* showModalDialog(...). *Such corner cases do not come into play very often, though.*

The disruptive, browser-blocking behavior of busy JavaScript loops requires the implementation of some mitigation on the browser level. We will explore these mitigations in detail in Chapter 14. For now, suffice it to say that they have another highly unusual consequence: Any endless loop may, in fact, terminate, in a fashion similar to throwing an unhandled exception. The engine will then return to the idle state but will remain operational, the offending code will remain callable, and all timers and event handlers will stay in place.

When triggered on purpose by the attacker, the ability to unexpectedly terminate the execution of CPU-intensive code may put the application in an inconsistent state by aborting an operation that the author expects to always complete successfully. And that's not all: Another, closely related consequence of these semantics should become evident in "JavaScript Object Notation and Other Data Serializations" on page 104.

Code and Object Inspection Capabilities

The JavaScript language has a rudimentary provision for inspecting the decompiled source code of any nonnative functions, simply by invoking the *toString()* or *toSource()* method on any function that the developer wishes to examine. Beyond that capability, opportunities to inspect the flow of programs are limited. Applications may leverage access to the in-memory representation of their host document and look up all inlined <*script*> blocks, but there is no direct visibility into any remotely loaded or dynamically generated code. Some insight into the call stack may also be gained through a nonstandard *caller* property, but there is also no way to tell which line of code is being currently executed or which one is coming up next.

The ability to dynamically create new JavaScript code is a more prominent part of the language. It is possible to instruct the engine to synchronously interpret strings passed to the built-in *eval(...)* function. For example, this will display an alert dialog:

```
eval("alert(\"Hi mom!\")")
```

Syntax errors in any input text provided to *eval(...)* will cause this function to throw an exception. Similarly, if parsing succeeds, any unhandled exceptions thrown by the interpreted code will be passed down to the caller. Finally, in the absence of syntax errors or runtime problems, the value of the last statement evaluated by the engine while executing the supplied code will be used as the return value of *eval(...)* itself.

In addition to this function, other browser-level mechanisms can be leveraged to schedule deferred parsing and execution of new JavaScript blocks once the execution engine returns to the idle state. Examples of such mechanisms include timers (*setTimeout*, *setInterval*), event handlers (*onclick*, *onload*, and so on), and interfaces to the HTML parser itself (*innerHTML*, *document.write(...)*, and such).

Whereas the ability to inspect the code is somewhat underhanded, runtime object introspection capabilities are well developed in JavaScript. Applications are permitted to enumerate almost any object method or property using simple *for ... in* or *for each ... in* iterators and can leverage operators such as *typeof*, *instanceof*, or "strictly equals" (===) and properties such as *length* to gain additional insight into the identity of every discovered item.

All of the foregoing features make it largely impossible for scripts running in the same context to keep secrets from each other. The functionality also makes it more difficult to keep secrets across document contexts, a problem that browser vendors had to combat for a very long time—and that, as you'll learn in Chapter 11, is still not completely a thing of the past.

Modifying the Runtime Environment

Despite the relative simplicity of the JavaScript language, executed scripts have many unusual ways of profoundly manipulating the behavior of their own JavaScript sandbox. In some rare cases, these behaviors can impact other documents, as well.

Overriding Built-Ins

One of the more unusual tools at the disposal of a rogue script is the ability to delete, overwrite, or shadow most of the built-in JavaScript functions and virtually all browser-supplied I/O methods. For example, consider the behavior of the following code:

```
// This assignment will not trigger an error.
eval = alert;

// This call will unexpectedly open a dialog prompt.
eval("Hi mom!");
```

And this is just where the fun begins. In Chrome, Safari, and Opera, it is possible to subsequently remove the *eval(...)* function altogether, using the *delete* operator. Confusingly, attempting the same in Firefox will restore the original built-in function, undoing the effect of the original override. Finally, in Internet Explorer, the deletion attempt will generate a belated exception that seems to serve no meaningful purpose at that point.

Further along these lines, almost every object, including built-ins such as *String* or *Array*, has a freely modifiable prototype. This prototype is a master object from which all existing and future object instances derive their methods and properties (forming a crude equivalent of class inheritance present in more fully featured programming languages). The ability to tamper with object prototypes can cause rather counterintuitive behavior of newly created objects, as illustrated here:

```
Number.prototype.toString = function() {
  return "Gotcha!";
};

// This will display "Gotcha!" instead of "42":
alert(new Number(42));
```

Setters and Getters

More interesting features of the object model available in contemporary dialects of JavaScript are *setters* and *getters*: ways to supply custom code that handles reading or setting properties of the host object. Although not as powerful as operator overloading in C++, these can be used to make existing objects or object prototypes behave in even more confusing ways. In the following snippet, the acts of setting the object property and reading it back later on are both subverted easily:

```
var evil_object = {
    set foo() { alert("Gotcha!"); },
    get foo() { return 2; }
  };

// This will display "Gotcha!" and have no other effect.
evil_object.foo = 1;

// This comparison will fail.
if (evil_object.foo != 1) alert("What's going on?!");
```

NOTE *Setters and getters were initially developed as a vendor extension but are now standardized under ECMAScript edition 5. The feature is available in all modern browsers but not in Internet Explorer 6 or 7.*

Impact on Potential Uses of the Language

As a result of the techniques discussed in the previous two sections, a script executing inside a context once tainted by any other untrusted content has no reliable way to examine its operating environment or take corrective

action; even the behavior of simple conditional expressions or loops can't necessarily be relied upon. The proposed enhancements to the language are likely to make the picture even more complicated. For example, the failed proposal for ECMAScript edition 4 featured full-fledged operator overloading, and this idea may return.

Even more interestingly, these design decisions also make it difficult to inspect any execution context from outside the per-page sandbox. For example, blind reliance on the reliability of the *location* object of a potentially hostile document has led to a fair number of security vulnerabilities in browser plug-ins, JavaScript-based extensions, and several classes of client-side web application security features. These vulnerabilities eventually resulted in the development of browser-level workarounds designed to partially protect this specific object against sabotage, but most of the remaining object hierarchy is up for grabs.

NOTE *The ability to tamper with one's own execution context is limited in the "strict" mode of ECMAScript edition 5. This mode is not fully supported in any browser as of this writing, however, and is meant to be an opt-in, discretionary mechanism.*

JavaScript Object Notation and Other Data Serializations

A very important syntax structure in JavaScript is its very compact and convenient in-place object serialization, known as JavaScript Object Notation, or JSON (RFC 4627[4]). This data format relies on overloading the meaning of the curly bracket symbol ({). When such a brace is used to open a fully qualified statement, it is treated in a familiar way, as the start of a nested code block. In an expression, however, it is assumed to be the beginning of a serialized object. The following example illustrates a correct use of this syntax and will display a simple prompt:

```
var impromptu_object = {
    "given_name"    : "John",
    "family_name"   : "Smith",
    "lucky_numbers" : [ 11630, 12067, 12407, 12887 ]
  };

// This will display "John".
alert(impromptu_object.given_name);
```

In contrast to the unambiguous serializations of numbers, strings, or arrays, the overloading of the curly bracket means that JSON blocks will not be recognized properly when used as a standalone statement. This may seem insignificant, but it is an advantage: It prevents any server-supplied responses that comply with this syntax from being meaningfully included across domains via <*script src*=...>.[*] The listing that follows will cause a syntax error, ostensibly

[*] Unlike most other content inclusion schemes available to scripts (such as *XMLHttpRequest*), <*script src*=...> is not subject to the cross-domain security restrictions outlined in Chapter 9. Therefore, the mechanism is a security risk whenever ambient authority credentials, such as cookies, are used by the server to dynamically generate user-specific JavaScript code. This class of vulnerabilities is unimaginatively referred to as *cross-site script inclusion*, or *XSSI*.

due to an illegal quote (❶) in what the interpreter attempts to treat as a code label,* and will have no measurable side effects:

```
<script>
{
❶  "given_name"    : "John",
  "family_name"   : "Smith",
  "lucky_numbers" : [ 11630, 12067, 12407, 12887 ]
};
</script>
```

NOTE *The inability to include JSON via* <script src=...> *is an interesting property, but it is also a fragile one. In particular, wrapping the response in parentheses or square brackets, or removing quotes around the labels, will render the syntax readily executable in a standalone block, which may have observable side effects. Given the rapidly evolving syntax of JavaScript, it is not wise to bank on this particular code layout always causing a parsing error in the years to come. That said, in many noncritical uses, this level of assurance will be good enough to rely on as a simple security mechanism.*

Once retrieved through a channel such as *XMLHttpRequest*, the JSON serialization can be quickly and effortlessly converted to an in-memory object using the *JSON.parse(...)* function in all common browsers, other than Internet Explorer. Unfortunately, for purposes of compatibility with Internet Explorer, and sometimes just out of custom, many developers resort to an equally fast yet far more dangerous hack:

```
var parsed_object = eval("(" + json_text + ")");
```

The problem with this syntax is that the *eval(...)* function used to compute the "value" of a JSON expression permits not only pure JSON inputs but any other well-formed JavaScript syntax to appear in the string. This can have undesirable, global side effects. For example, the function call embedded in this faux JSON response will execute:

```
{ "given_name": alert("Hi mom!") }
```

This behavior creates an additional burden on web developers to accept JSON payloads only from trusted sources and always to correctly escape feeds produced by their own server-side code. Predictably, failure to do so has contributed a fair number of application-level security bugs.

NOTE *The difficulty of getting* eval(...) *right is embodied by the JSON specification (RFC 4627) itself: The allegedly secure parser implementation included in that document unintentionally permits rogue JSON responses to freely increment or decrement any program variables that happen to consist solely of the letters "a", "e", "f", "l", "n", "r",*

* Somewhat unexpectedly, JavaScript supports C-style labeled statements, such as *my_label: alert("Hi mom!")*. This is interesting because for philosophical reasons, the language has no support for *goto* and, therefore, such a label can't be meaningfully referenced in most cases.

"s", "t", "u", plus digits; that's enough to spell "unsafe" and about 1,000 other common English words. The faulty regular expression legitimized in this RFC appears all over the Internet and will continue to do so.

Thanks to their ease of use, JSON serializations are ubiquitous in server-to-client communications across all modern web applications. The format is rivaled only by other, less secure string or array serializations and by JSONP.* All of these schemes are incompatible with *JSON.parse(…)*, however, and must rely on unsafe *eval(…)* to be converted to in-memory data. The other property of these formats is that, unlike proper JSON, they will parse properly when loaded with *<script src=…>* on a third-party page. This property is advantageous in some rare cases, but mostly it just constitutes an unobvious risk. For example, consider that even though loading an array serialization via a *<script>* tag normally has no measurable side effects, an attacker could, at least until recent improvements, modify the setters on an *Array* prototype to retrieve the supplied data. A common but often insufficient practice of prefixing a response with a *while(1);* loop to prevent this attack can backfire in interesting ways if you recall the possibility of endless loops terminating in JavaScript.

E4X and Other Syntax Extensions

Like HTML, JavaScript is quickly evolving. Some of the changes made to it over the years have been fairly radical and may end up turning text formats that were previously rejected by the parser into a valid JavaScript code. This, in turn, may lead to unexpected data disclosure, especially in conjunction with the extensive code and object inspection and modification capabilities discussed earlier in this chapter—and the ability to use *<script src=…>* to load cross-domain code.

One of the more notable examples of this trend is *ECMAScript for XML* (E4X),[5] a completely unnecessary but elegant plan to incorporate XML syntax directly into JavaScript as an alternative to JSON-style serializations. In any E4X-compatible engine, such as Firefox, the following two snippets of code would be roughly equivalent:

```
// Normal object serialization
var my_object = { "user": {
                  "given_name": "John",
                  "family_name": "Smith",
                  "id": make_up_value()
                } };

// E4X serialization
var my_object = <user>
                  <given_name>John</given_name>
                  <family_name>Smith</family_name>
                  <id>{ make_up_value() }</id>
                </user>;
```

*JSONP literally means "JSON with padding" and stands for JSON serialization wrapped in some supplementary code that turns it into a valid, standalone JavaScript statement for convenience. Common examples may include a function call (e.g., *callback_function({ …JSON data… })*) or a variable assignment (*var return_value = { …JSON data… }*).

The unexpected consequence of E4X is that, under this regime, any well-formed XML document suddenly becomes a valid *<script src=…>* target that will parse as an expression-as-statement block. Moreover, if an attacker can strategically place "{" and "}" characters on an included page, or alter the setters for the right object prototype, the attacker may be able to extract user-specific text displayed in an unrelated document. The following example illustrates the risk:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  ...
  { steal_stuff(                                  <-- attacker-supplied string
    ...
    <span>User-specific secrets here</span>
    ...
  ) }                                             <-- attacker-supplied string
  ...
</html>
```

To their credit, after several years of living with the flaw, Firefox developers decided to disallow any E4X statements that span the entirety of any parsed script, partly closing this loophole. Nevertheless, the fluidity of the language is evident, and it casts some doubt on the robustness of using of JSON responses as a defense against cross-domain script inclusion. The moment a third meaning is given to the "{" symbol or quotes-as-labels start having a purpose, the security of this server-to-client data exchange format will be substantially degraded. Be sure to plan ahead.

Standard Object Hierarchy

The JavaScript execution environment is structured around an implicit root object, which is used as the default namespace for all global variables and functions created by the program. In addition to a handful of language-mandated built-ins, this namespace is prepopulated with a hierarchy of functions that implement input and output capabilities in the browser environment. These capabilities include manipulating browser windows (*open(…)*, *close()*, *moveTo(…)*, *resizeTo(…)*, *focus()*, *blur()*, and such); configuring JavaScript timers (*setTimeout(…)*, *setInterval(…)*, and so on); displaying various UI prompts (*alert(…)*, *prompt(…)*, *print(…)*); and performing a variety of other vendor-specific and frequently risky functions, such as accessing the system clipboard, creating bookmarks, or changing the home page.

The top-level object also provides JavaScript references to root objects belonging to related contexts, including the parent frame (*parent*), the top-level document in the current browser window (*top*), the window that created the current one (*opener*), and all subframes of the current document (*frames[]*). Several circular references to the current root object itself are also included—say, *window* and *self*. In browsers other than Firefox, elements with specified *id* or *name* parameters will be automatically registered in this namespace, too, permitting syntax such as this:

```
<img id="hello" src="http://www.example.com/">
...
```

```
<script>
  alert(hello.src);
</script>
```

Thankfully, in case of any name conflicts with JavaScript variables or built-ins, *id* data will not be given precedence, largely avoiding any possible interference between otherwise sanitized, user-supplied markup and in-document scripts.

The remainder of the top-level hierarchy consists primarily of a couple of distinguished children objects that group browser API features by theme:

***location* object**
: This is a collection of properties and methods that allow the program to read the URL of the current document or initiate navigation to a new one. This last action, in most cases, is lethal to the caller: The current scripting context will be destroyed and replaced with a new one shortly thereafter. Updating just the fragment identifier (*location.hash*) is an exception to this rule, as explained in Chapter 2.

  Note that when using *location.** data to construct new strings (HTML and JavaScript code in particular), it is unsafe to assume that it is escaped in any specific way. Internet Explorer will keep angle brackets as is in the *location.search* property (which corresponds to the URL query string). Chrome, on the other hand, will escape them, but it will glance over double quotes (") or backslashes. Most browsers also do not apply any escaping to the fragment ID.

***history* object**
: This hierarchy provides several infrequently used methods for moving through the per-window browsing history, in a manner similar to clicking the "back" and "forward" buttons in the browser UI. It is not possible to directly examine any of the previously visited URLs; the only option is to navigate to the history blindly by providing numerical offsets, such as *history.go(-2)*. (Some recent additions to this hierarchy will be discussed in Chapter 17.)

***screen* object**
: A basic API for examining the dimensions of the screen and the browser window, monitor DPI, color depth, and so on. This is offered to help websites optimize the presentation of a page for a particular display device.

***navigator* object**
: An interface for querying the browser version, the underlying operating system, and the list of installed plug-ins.

***document* object**
: By far the most complex of the hierarchies, this is a doorway to the Document Object Model[6] of the current page; we will have a look at this model in the following section. A couple of functions not related to document structure also appear under the *document* hierarchy, usually due to arbitrary design decisions. Examples include *document.cookie* for manipulating cookies, *document.write(…)* for appending HTML to the current page, and *document.execCommand(…)* for performing certain WYSIWYG editing tasks.

NOTE *Interestingly, the information available through the* navigator *and* screen *objects is sufficient to uniquely fingerprint many users with a high degree of confidence. This long-known property is emphatically demonstrated by* Panopticlick*, a project of the Electronic Frontier Foundation:* https://panopticlick.eff.org/.

Several other language-mandated objects offer simple string-processing or arithmetic capabilities. For example, *Math.random()* implements an unsafe, predictable pseudo-random number generator (a safe PRNG alternative is unfortunately not available at this time in most browsers[*]), while *String.fromCharCode()* can be used to convert numerical values into Unicode strings. In privileged execution contexts, which are not reachable by normal web applications, a fair number of other task-specific objects will also appear.

NOTE *When accessing any of the browser-supplied objects, it is important to remember that while JavaScript does not use NUL-terminated ASCIZ strings, the underlying browser (written in C or C++) sometimes will. Therefore, the outcomes of assigning NUL-containing strings to various DOM properties, or supplying them to native functions, may be unpredictable and inconsistent. Almost all browsers truncate assignments to* location.* *at NUL, but only some engines will do the same when dealing with DOM* *.innerHTML.

The Document Object Model

The Document Object Model, accessible through the *document* hierarchy, provides a structured, in-memory representation of the current document as mapped out by the HTML parser. The resulting object tree exposes all HTML elements on the page, their tag-specific methods and properties, and the associated CSS data. This representation, not the original HTML source, is used by the browser to render and update the currently displayed document.

JavaScript can access the DOM in a very straightforward way, similarly to any normal objects. For example, the following snippet will go to the fifth tag within the document's <*body*> block, look up the first nested subtag, and set that element's CSS color to red:

```
document.body.children[4].children[0].style.color = "red";
```

To avoid having to waddle through the DOM tree in order to get to a particular deeply nested element, the browser provides several document-wide lookup functions, such as *getElementById(…)* and *getElementsByTagName(…)*, as well as partly redundant grouping mechanisms such as *frames[]*, *images[]*, or *forms[]*. These features permit syntax such as the following two lines of code, both of which directly reference an element no matter where in the document hierarchy it happens to appear:

```
document.getElementsByTagName("input")[2].value = "Hi mom!";
document.images[7].src = "/example.jpg";
```

[*] There are a recently added *window.crypto.getRandomValues(…)* API in Chrome and a currently nonoperational *window.crypto.random(…)* API in Firefox.

For legacy reasons, the names of certain HTML elements (<*img*>, <*form*>, <*embed*>, <*object*>, and <*applet*>) are also directly mapped to the *document* namespace, as illustrated in the following snippet:

```
<img name="hello" src="http://www.example.com/">

<script>
  alert(document.hello.src);
</script>
```

Unlike in the more reasonable case of *name* and *id* mapping in the global namespace (see previous section), such *document* entries may clobber built-in functions and objects such as *getElementById* or *body*. Therefore, permitting user-specified tag names, for example for the purpose of constructing forms, can be unsafe.

In addition to providing access to an abstract representation of the document, many DOM nodes may expose properties such as *innerHTML* and *outerHTML*, which permit a portion of the document tree to be read back as a well-formed, serialized HTML string. Interestingly, the same property can be written to in order to replace any portion of the DOM tree with the result of parsing a script-supplied snippet of HTML. One example of that last use is this:

```
document.getElementById("output").innerHTML = "<b>Hi mom!</b>";
```

Every assignment to *innerHTML* must involve a well-formed and self-contained block of HTML that does not alter the document hierarchy outside the substituted fragment. If this condition is not met, the input will be coerced to a well-formed syntax before the substitution takes place. Therefore, the following example will not work as expected; that is, it will not display "Hi mom!" in bold and will not put the remainder of the document in italics:

```
some_element.innerHTML  = "<b>Hi";
some_element.innerHTML += " mom!</b><i>";
```

Instead, each of these two assignments will be processed and corrected individually, resulting in a behavior equivalent to this:

```
some_element.innerHTML = "<b>Hi</b> mom!<i></i>";
```

It is important to note that the *innerHTML* mechanism should be used with extreme caution. In addition to being inherently prone to markup injection if proper HTML escaping is not observed, browser implementations of the DOM-to-HTML serialization algorithms are often imperfect. A recent (now fixed) example of such a problem in WebKit[7] is illustrated here:

```
<textarea>
  &lt;/textarea&gt;&lt;script&gt;alert(1)&lt;/script&gt;
</textarea>
```

Because of the confusion over the semantics of *<textarea>*, this seemingly unambiguous input markup, when parsed to a DOM tree and then accessed through *innerHTML*, would be incorrectly read back as:

```
<textarea>
  </textarea><script>alert(1)</script>
</textarea>
```

In such a situation, even performing a no-op assignment of this serialization (such as *some_element.innerHTML += ""*) would lead to unexpected script injection. Similar problems tend to plague other browsers, too. For example, Internet Explorer developers working on the *innerHTML* code were unaware that MSHTML recognizes backticks (`) as quote characters and so ended up handling them incorrectly. In their implementation, the following markup:

```
<img src="test.jpg" alt="``onload=alert(1)">
```

would be reserialized as this:

```
<img src=test.jpg alt=``onload=alert(1)>
```

Individual bugs aside, the situation with *innerHTML* is pretty dire: Section 10.3 of the current draft of HTML5 simply acknowledges that certain script-created DOM structures are completely impossible to serialize to HTML and does not require browsers to behave sensibly in such a case. *Caveat emptor!*

Access to Other Documents

Scripts may come into possession of object handles that point to the root hierarchy of another scripting context. For example, by default, every context can readily reference *parent*, *top*, *opener*, and *frames[]*, all supplied to it in the top-level object. Calling the *window.open(...)* function to create a new window will also return a reference, and so will an attempt to look up an existing named window using this syntax:

```
var window_handle = window.open("", "window_name");
```

Once the program holds a handle pointing to another scripting context, it may attempt to interact with that context, subject to security checks discussed in Chapter 9. An example of a simple interaction might be as follows:

```
top.location.path = "/new_path.html";
```

or

```
frames[2].document.getElementById("output").innerHTML = "Hi mom!";
```

In the absence of a valid handle, JavaScript-level interaction with an unrelated document should not be possible. In particular, there is no way to look up unnamed windows opened in completely separate navigation flows, at least until their name is explicitly set by one of the visited pages (the *window.name* property permits this).

Script Character Encoding

JavaScript engines support several familiar, backslash-based string-encoding methods that can be employed to escape quote characters, HTML markup, and other problematic bits in the embedded text. These methods are as follows:

- C-style shorthand notation for certain control characters: *\b* for backspace, *\t* for horizontal tab, *\v* for vertical tab, *\f* for form feed, *\r* for CR, and *\n* for LF. This exact set of escape codes is recognized by both ECMAScript and the JSON RFC.
- Three-digit, zero-padded, 8-bit octal character codes with no prefix (such as "\145" instead of "e"). This C-inspired syntax is not a part of ECMAScript but is in practice supported by all scripting engines, both in normal code and in *JSON.parse(...)*.
- Two-digit, zero-padded, 8-bit hexadecimal character codes, prefixed with "x" ("e" becomes "\x65"). Again, this scheme is not endorsed by ECMAScript or RFC 4627, but having its roots in the C language, it is widely supported in practice.
- Four-digit, zero-padded, 16-bit hexadecimal Unicode values, prefixed with "u" ("e" turns into "\u0065"). This format is sanctioned by ECMAScript and RFC 4627 and is supported by all modern browsers.
- A backslash followed by any character other than an octal digit; "b", "t", "v", "f", "r," or "n" characters used for other predefined escape sequences; and "x" or "u". In this scheme, the subsequent character will be treated as a literal. ECMAScript permits this scheme to be used to escape only quotes and the backslash character itself, but in practice, any other value is accepted as well.

  This approach is somewhat error prone, and as in the case of CSS, it should not be used to escape angle brackets and other HTML syntax delimiters. This is because JavaScript parsing takes place after HTML parsing, and the backslash prefix will be not treated in any special way by the HTML parser itself.

NOTE *Somewhat inexplicably, Internet Explorer does not recognize the vertical tab ("\v") shorthand, thereby creating one of the more convenient (but very naughty!) ways to test for that particular browser:*

```
if ("\v" == "v") alert("Looks like Internet Explorer!");
```

Surprisingly, the Unicode-based escaping method (but not the other ones) is also recognized outside strings. Although the idea seems arbitrary, the behavior is a bit more sensible than with CSS: Escape codes can be used only in identifiers, and they will not work as a substitute for any syntax-sensitive symbols. Therefore, the following is possible:

```
\u0061lert("This displays a message!");
```

On the other hand, any attempt to substitute the parentheses or quotes in a similar fashion would fail.

Unlike in some C or C++ implementations, stray multiline string literals are not tolerated by any JavaScript engine. That said, despite a strongly worded prohibition in ECMAScript specs, there is one exception: A lone backslash at the end of a line may be used to join multiline literals seamlessly. This behavior is illustrated below:

```
var text = 'This syntax
            is invalid.';

var text = 'This syntax, on the other hand, \
            is OK in all browsers.';
```

Code Inclusion Modes and Nesting Risks

As should be evident from the earlier discussions in this chapter, there are several ways to execute scripts in the context of the current page. It is probably useful to enumerate some of the most common ones:

- Inline *<script>* blocks
- Remote scripts loaded with *<script src=...>*[*]
- *javascript:* URLs in various HTML parameters and in CSS
- CSS *expression(...)* syntax and XBL bindings in certain browsers
- Event handlers (*onload*, *onerror*, *onclick*, etc.)
- Timers (*setTimeout*, *setInterval*)
- *eval(...)* calls

Combining these methods often seems natural, but doing so can create very unexpected and dangerous parsing chains. For example, consider the transformation that would need to be applied to the value inserted by the server in place of *user_string* in this code:

```
<div onclick="setTimeout('do_stuff(\'user_string\')', 1000)">
```

[*] On both types of *<script>* blocks, Microsoft supports a pseudo-dialect called *JScript.Encode*. This mode can be selected by specifying a *language* parameter on the *<script>* tag and simply permits the actual script to be encoded using a trivial alphabet substitution cipher to make it unreadable to casual users. The mechanism is completely worthless from the security standpoint, as the "encryption" can be reverted easily.

It is often difficult to notice that the value will go through no fewer than three rounds of parsing! First, the HTML parser will extract the *onclick* parameter and put it into DOM; next, when the button is clicked, the first round of JavaScript parsing will extract the *setTimeout(...)* syntax; and finally, one second after the initial click, the actual *do_stuff(...)* sequence will be parsed and executed.

Therefore, in the example above, in order to survive the process, *user_string* needs to be double-encoded using JavaScript backslash sequences, and then encoded again using HTML entities, in that exact order. Any different approach will likely lead to code injection.

Another tricky escaping situation is illustrated here:

```
<script>
var some_value = "user_string";
...
setTimeout("do_stuff('" + some_value + "')", 1000);
</script>
```

Even though the initial assignment of *some_value* requires *user_string* to be escaped just once, the subsequent ad hoc construction of a second-order script in the *setTimeout(...)* parameter introduces a vulnerability if no additional escaping is applied beforehand.

Such coding patterns happen frequently in JavaScript programs, and they are very easy to miss. It is much better to consistently discourage them than to audit the resulting code.

The Living Dead: Visual Basic

Having covered most of the needed ground related to JavaScript, it's time for an honorable mention of the long-forgotten contender for the scripting throne. Despite 15 years of lingering in almost complete obscurity, browser-side VBScript is still supported in Internet Explorer. In most aspects, Microsoft's language is supposed to be functionally equivalent to JavaScript, and it has access to exactly the same Document Object Model APIs and other built-in functions as JavaScript. But, as one might expect, some tweaks and extensions are present—for example, a couple of VB-specific functions in place of the JavaScript built-ins.

There is virtually no research into the security properties of VBScript, the robustness of the parser, or its potential incompatibilities with the modern DOM. Anecdotal evidence suggests that the language receives no consistent scrutiny on Microsoft's end, either. For example, the built-in *MsgBox*[8] can be used to display modal, always-on-top prompts with a degree of flexibility completely unheard of in the JavaScript world, leaving *alert(...)* in the dust.

It is difficult to predict how long VBScript will continue to be supported in this browser and what unexpected consequences for user and web application security it is yet to have. Only time will tell.

Security Engineering Cheat Sheet

When Loading Remote Scripts

As with CSS, you are linking the security of your site to the originating domain of the script. When in doubt, make a local copy of the data instead. On HTTPS sites, require all scripts to be served over HTTPS.

When Parsing JSON Received from the Server

Rely on *JSON.parse(…)* where supported. Do not use *eval(…)* or the *eval*-based implementation provided in RFC 4627. Both are unsafe, especially when processing data from third parties. A later implementation from the author of RFC 4627, *json2.js*,[9] is probably okay.

When Putting User-Supplied Data Inside JavaScript Blocks

- ☑ **Stand-alone strings in <*script*> blocks:** Backslash-escape all control characters (0x00–0x1F), "\", "<", ">", and quotes using numerical codes. It is also preferable to escape high-bit characters.

  Do not rely on user-supplied strings to construct dynamic HTML. Always use safe DOM features such as *innerText* or *createTextNode(…)* instead. Do not use user-supplied strings to construct second-order scripts; avoid *eval(…)*, *setTimeout(…)*, and so on.
- ☑ **Stand-alone strings in separately served scripts:** Follow the same rules as for <*script*> blocks. If your scripts contain any sensitive, user-specific information, be sure to account for cross-site script inclusion risks; use reliable parser-busting prefixes, such as ")}]'\n", near the beginning of a file or, at the very minimum, use a proper JSON serialization with no padding or other tweaks. Additionally, consult Chapter 13 for tips on how to prevent cross-site scripting in non-HTML content.
- ☑ **Strings in inlined event handlers, *javascript:* URLs, and so on:** Multiple levels of escaping are involved. Do not attempt this because it is error prone. If unavoidable, apply the above JS escaping rules first and then apply HTML or URL parameter encoding, as applicable, to the resulting string. Never use in conjunction with *eval(…)*, *setTimeout(…)*, *innerHTML*, and such.
- ☑ **Nonstring content:** Allow only whitelisted alphanumeric keywords and carefully validated numerical values. Do not attempt to reject known bad patterns instead.

When Interacting with Browser Objects on the Client Side

- ☑ **Generating HTML content on the client side:** Do not resort to *innerHTML*, *document.write(…)*, and similar tools because they are prone to introducing cross-site scripting flaws, often in unexpected ways. Use safe methods such as *createElement(…)* and *appendChild(…)* and properties such as *innerText* or *textContent* to construct the document instead.
- ☑ **Relying on user-controlled data:** Make no assumptions about the escaping rules applied to any values read back from the browser and, in particular, to *location* properties and other external sources of URLs, which are inconsistent and vary from one implementation to another. Always do your own escaping.

If You Want to Allow User-Controlled Scripts on Your Page

It is virtually impossible to do this safely. Experimental JavaScript rewriting frameworks, such as Caja (*http://code.google.com/p/google-caja/*), are the only portable option. Also see Chapter 16 for information on sandboxed frames, an upcoming alternative for embedding untrusted gadgets on web pages.

7

NON-HTML DOCUMENT TYPES

In addition to HTML documents, about a dozen other file formats are recognized and displayed by the rendering engines of modern web browsers; a list that is likely to grow over time.

Because of the powerful scripting capabilities available in some of these formats, and because of the antics of browser-content handling, the set of natively supported non-HTML inputs deserves a closer examination at this point, even if a detailed discussion of some of their less-obvious security consequences—such as *content sniffing*—will have to wait until Part II of this book.

Plaintext Files

Perhaps the most prosaic type of non-HTML document recognized by every single browser is a plaintext file. In this rendering mode, the input is simply displayed as is, typically using a nonproportional typeface, and save for optional character set transcoding, the data is not altered in any way.

All browsers recognize plaintext files served with *Content-Type: text/plain* in the HTTP headers. In all implementations but Internet Explorer, plaintext is also the fallback display method for headerless HTTP/0.9 responses and HTTP/1.*x* data with *Content-Type* missing; in both these cases, plaintext is used when all other content detection heuristics fail. (Internet Explorer unconditionally falls back to HTML rendering, true to the letter of Tim Berners-Lee's original protocol drafts.)

For the convenience of developers, most browsers also automatically map several other MIME types, including *application/javascript* and friends[*] or *text/css*, to plaintext. Interestingly, *application/json*, the value mandated for JSON responses in RFC 4627, is not on the list (perhaps because it is seldom used in practice).

Plaintext rendering has no specific security consequences. That said, due to a range of poor design decisions in other browser components and in third-party code, even seemingly harmless non-HTML formats are at a risk of being misidentified as, for example, HTML. Attacker-controlled plaintext documents are of special concern because their layout is often fairly unconstrained and therefore particularly conducive to being misidentified. Chapter 13 dissects these threats and provides advice on how to mitigate the risk.

Bitmap Images

Browser-rendering engines recognize direct navigation to the same set of bitmap image formats that are normally supported in HTML documents when loaded via the <*img*> tag, including JPEG, PNG, GIF, BMP, and a couple more. When the user navigates directly to such a resource, the decoded bitmap is shown in the document window, allowing the user little more than the ability to scroll, zoom in and out, and save the file to disk.

In the absence of *Content-Type* information, images are detected based on file header checks. When a *Content-Type* value is present, it is compared with about a dozen predefined image types, and the user is routed accordingly. But if an attempt to decode the image fails, file headers are used to make a second guess. It is therefore possible (but, for the reasons explored in Chapter 13, often unwise) to serve a GIF file as *image/jpeg*.

As with text files, bitmap images are a passive resource and carry no unusual security risks.[†] However, whenever serving user-supplied images, remember that attackers will have a degree of control over the data, even if the format is carefully validated and scaled or recompressed. Therefore, the concerns about such a document format being misinterpreted by a browser or a plug-in still remain.

[*] The official MIME type for JavaScript is *application/javascript*, as per RFC 4329, but about a dozen other values have been used in the past (e.g., *text/javascript, application/x-javascript, application/ecmascript*).

[†] Naturally, exploitable coding errors occasionally happen in all programs that deal with complex data formats, and image parsers are no exception.

Audio and Video

For a very long time, browsers had no built-in support for playing audio and video content, save for an obscure and oft-ridiculed *<bgsound>* tag in Internet Explorer, which to this day can be used to play simple MID or WAV files. In the absence of real, cross-browser multimedia playback functionality, audio and video were almost exclusively the domain of browser plug-ins, whether purpose-built (such as Windows Media Player or Apple QuickTime) or generic (Adobe Flash, Microsoft Silverlight, and so on).

The ongoing work on HTML5 seeks to change this through support for *<audio>* and *<video>* tags: convenient, scriptable methods to interface with built-in media decoders. Unfortunately, there is substantial vendor-level disagreement as to which video formats to support and what patent consequences this decision may have. For example, while many browsers already support Ogg Theora (a free, open source, but somewhat niche codec), spirited arguments surrounding the merits of supporting the very popular but patent- and royalty-encumbered H.264 format and the prospects of a new, Google-backed WebM alternative will probably continue for the foreseeable future.

As with other passive media formats (and unlike some types of plug-in-rendered content!), neither *<bgsound>* nor HTML5 multimedia are expected to have any unusual implications for web application security, as long as the possibility of content misidentification is mitigated appropriately.[*]

XML-Based Documents

Readers who found the handling of the formats discussed so far to be too sane for their tastes are in for a well-deserved treat. The largest and definitely most interesting family of browser-supported non-HTML document types relies on the common XML syntax and provides more than a fair share of interesting surprises.

Several of the formats belonging to this category are forwarded to specialized, single-purpose XML analyzers, usually based on the received *Content-Type* value or other simple heuristics. But more commonly, the payload is routed to the same parser that is relied upon to render XHTML documents and then displayed using this common pipeline.

In the latter case, the actual meaning of the document is determined by the URL-like *xmlns* namespace directives present in the markup itself, and the namespace parameter may have nothing to do with the value originally supplied in *Content-Type*. Quite simply, there is no mechanism that would prevent a document served as *application/mathml+xml* from containing nothing but XHTML markup and beginning with *<html xmlns="http://www.w3.org/1999/xhtml">*.

[*] But some far-fetched interactions between various technologies are a distinct possibility. For example, what if the *<audio>* tag supports raw, uncompressed audio and is pointed to a sensitive nonaudio document, and then the proposed HTML5 microphone API is used by another website to capture the resulting waveform and reconstruct the contents of the file?

In the most common scenario, the namespace for the entire XML file is defined only once and is attached to the top-level tag. In principle, however, any number of different *xmlns* directives may appear in a single file, giving different meanings to each section of the document. For example:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <u>Hello world!</u>
  <svg xmlns="http://www.w3.org/2000/svg">
    <line x1="0" y1="0" x2="100" y2="100" style="stroke: red" />
  </svg>
</html>
```

Faced with such input, the general-purpose renderer will usually do its best to make sense of all the recognized namespaces and assemble the markup into a single, consistent document with a normal Document Object Model representation. And, if any one of the recognized namespaces happens to support scripting, any embedded scripts will execute, too.

Because of the somewhat counterintuitive *xmlns* handling behavior, *Content-Type* is not a suitable way to control how a particular XML document will be parsed; the presence of a particular top-level *xmlns* directive is also not a guarantee that no other data formats will be honored later on. Any attacker-controlled XML-based formats must therefore be handled with care and sanitized very thoroughly.

Generic XML View

In most browsers, a valid XML document with no renderer-recognized namespaces present anywhere in the markup will be shown as an interactive, pretty-printed representation of the document tree, as shown in Figure 7-1. This mode is not particularly useful to end users, but it can aid debugging.

That said, when any of the namespaces in the document is known to the browser (even when the top-level one is not recognized at all!), the document will be rendered differently: All recognized markup will work as intended, all unsupported tags will simply have no effect, and any text between them will be shown as is.

To illustrate this rendering strategy, consider the following input:

```
<foo xmlns="http://www.example.com/nonexistent">
     <u>Hello</u>
     <html xmlns="http://www.w3.org/1999/xhtml">
     <u>world!</u>
  </html>
</foo>
```

The above example will be rendered as "Hello world!" The first *<u>* tag, with no semantics-defining namespace associated with it, will have no visible effect. The second one will be understood as an XHTML tag that triggers underlining.

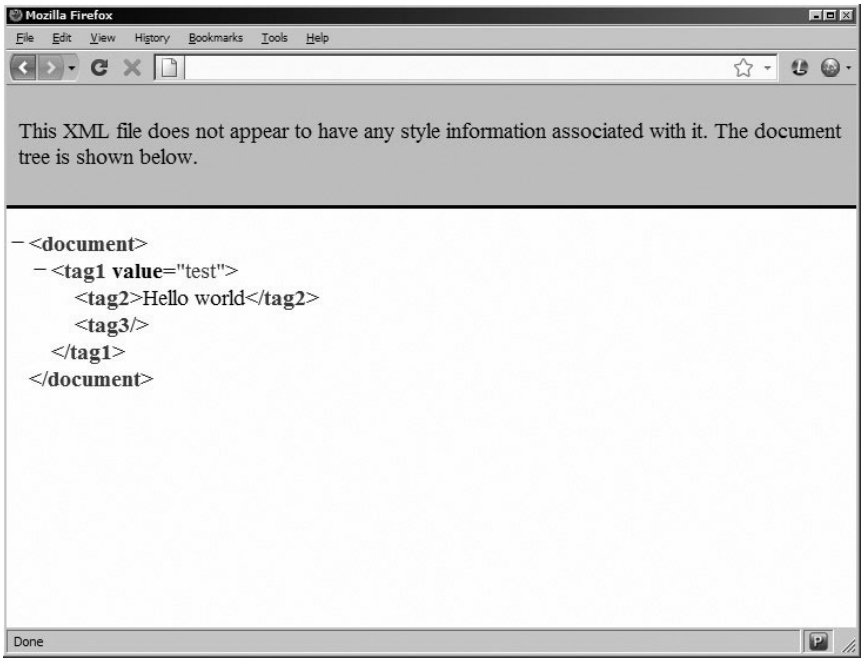


Figure 7-1: Firefox displaying an XML document with no recognized namespaces

The consequences of this fault-tolerant approach to the rendering of unknown XML documents and unrecognized namespaces are subtle but fairly important. For example, it will not be safe to proxy an unsanitized RSS feed, even though this format is typically routed to a specialized renderer and thus not subject to XSS risks. Any browser with no built-in RSS reader may fall back to generic rendering and then find HTML buried deep inside the feed.

Scalable Vector Graphics

Scalable Vector Graphics (SVG)[1] is a quickly evolving, XML-based vector graphics format. First published in 2001 by W3C, it is noteworthy for its integrated animation capabilities and direct JavaScript scripting features. The following example of a vector image draws a circle and displays a message when this circle is clicked:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <script><![CDATA[
    function clicked() { alert("Hi mom!"); }
  ]]></script>
  <circle onclick="clicked()" cx="50" cy="50"
          r="50" fill="pink" />
</svg>
```

The SVG file format is recognized all modern browsers except for Internet Explorer prior to 9, and it is handled by the general-purpose XML renderer. SVG images can be embedded into XHTML with an appropriate *xmlns* directive or inlined in non-XML HTML5 documents using a predefined <*svg*> tag.

Interestingly, in several browsers the format can also be placed in a standalone XML document and then viewed directly, or it can be loaded on third-party pages via the <*img*> markup. While it is safe to load SVG images via <*img*> (scripting should be disabled in this scenario), it is fairly dangerous to host user-supplied SVG data because in cases of direct navigation, all embedded scripts will execute in the context of the hosting domain. This unexpected problem means that serving any externally originating SVG images will require very careful syntax sanitization to eliminate non-SVG *xmlns* content from the XML container and to permit only certain types of markup in the remainder of the document.

NOTE *The* Content-Disposition *header on the relevant HTTP responses is a potential workaround that permits SVG to be included via* <img> *but not accessed directly. This approach is not perfect, but it limits the risk. Using a throwaway domain to host such images is another possibility.*

Mathematical Markup Language

Mathematical Markup Language (MathML)[2] is a fairly straightforward means to facilitate the semantic, if a bit verbose, representation of mathematical equations. The standard was originally proposed by the W3C in 1998, and it has been substantially refined through the years. Because of its somewhat niche application, MathML needed over a decade to gain partial support in Opera and Firefox browsers, but it is slowly gaining acceptance today. In the browsers that support the language, it may be placed in a standalone file or inline in XHTML and HTML5 documents.

Unlike SVG, MathML has no additional security considerations beyond those associated with generically handled XML.

XML User Interface Language

The XML User Interface Language (XUL)[3] is a presentation markup language created by Mozilla specifically for building browser-based applications, rather than documents. XUL exists because although modern HTML is often powerful enough to build basic graphical user interfaces, it is not particularly convenient for certain specialized tasks that desktop applications excel in, such as implementing common dialog windows or system menus.

XUL is not currently supported by any browser other than Firefox and appears to be disabled in the recent release, Firefox 6. In Firefox, it is handled by the general-purpose renderer, based on the appropriate *xmlns* namespace. Firefox uses XUL for much of its internal UI, but otherwise the language is seldom encountered on the Internet.

From the standpoint of web application security, Internet-originating XUL documents can be considered roughly equivalent to HTML documents.

Essentially, the language has JavaScript scripting capabilities and allows broad control over the appearance of the rendered page. Other than that property, it has no unusual quirks.

Wireless Markup Language

Wireless Markup Language (WML)[4] is a largely obsolete "optimized" HTML syntax developed in the 1990s by a consortium of mobile handset manufacturers and cellular network operators. This XML-based language, a part of the Wireless Application Protocol suite (WAP), offered a simplified weblike browsing experience for pre-smartphone devices with limited bandwidth and CPU resources.* A simple WML page might have looked like this:

```
<wml>
  <card title="Hello world!">
      <a href="elsewhere.wml">Click here!</a>
  </card>
</wml>
```

Because WAP services needed to be engineered independently of normal HTML content and had to deal with closed and underspecified client architectures and other carrier-imposed restrictions, WML never became as popular as its proponents hoped. In almost all developed markets, WML has been displaced by fast, Internet-enabled smartphones with fully featured HTML browsers. Nevertheless, the legacy of the language lives on, and it is still routed to specialized renderers in Opera and in Internet Explorer Mobile.

In the browsers that support the format, it is often possible to use WML-based scripts. There are two methods to achieve this. The canonical way is to use WMLScript (WMLS), a JavaScript-derived execution environment that depends on stand-alone script files, coupled with an extremely inconsiderate abuse of fragment IDs for an equivalent of possibly attacker-controlled *eval(…)* statements:

```
<a href="scriptfile.wmls#some_function()">Click here!</a>
```

The other method of executing scripts, available in more featured browsers, is to simply embed normal *javascript:* URLs or insert *<script>* blocks into the WML file.

RSS and Atom Feeds

Feeds are a standardized way for clients to periodically poll sites of interest to users (such as their favorite blogs) for machine-readable updates to said sites' content. Really Simple Syndication (RSS)[5] and Atom[6] are two superficially similar but fiercely competing XML-based feed formats. The first (RSS) is popular; the second (Atom) is said to be good.

* Astute readers will note that XML is not a particularly good way to conserve bandwidth or CPU resources. To that effect, the WAP suite provides an alternative, binary-only serialization of XML, known as WBXML.

Built-in, specialized RSS and Atom renderers are available in Firefox, Safari, and Opera. The determination to route an XML document to these modules is based on simple, browser-specific heuristics, such as the top-level tag being named *<rss>* or *<feed>* (and not having any conflicting *xmlns* directives). In Firefox, RSS parsing may kick in even if *Content-Type* is *image/svg+xml* or *text/html*. Safari will happily recognize feeds in even more unrelated MIME types.

One interesting feature of both feed formats is that they permit a subset of HTML, including CSS, to be embedded in a document in a rather peculiar, indirect way: as an entity-escaped text. Here is an example of this syntax:

```
<rss>
  ...
  <description type="html">
    &lt;u&gt; Underlined text! &lt;/u&gt;
  </description>
  ...
</rss>
```

The subset of HTML permitted in RSS and Atom feeds is not well defined, and some feed renderers have previously permitted direct scripting or navigation to potentially dangerous pseudo-URLs. Perhaps more importantly, however, any browser that does not have built-in feed previews may render the file using the generic XML parsing approach; if such feeds are not sanitized carefully, script execution will ensue.

A Note on Nonrenderable File Types

For the sake of completeness, it should be noted that all modern browsers support a number of specialized file formats that remain completely opaque to the renderer or to the web application layer but that are nevertheless recognized by a variety of in-browser subsystems.

A detailed investigation of these formats is beyond the scope of this book, but some notable examples include plug-in and extension installation manifests, automatic HTTP proxy autoconfiguration files (PAC), installable visual skins, Certificate Revocation Lists (CRLs), antimalware site blacklists, and downloadable TrueType and OpenType fonts.

The security properties of these mechanisms should be studied individually before deciding to allow any of these formats to be served to the user. Save for the generic content-hosting considerations outlined in Chapter 13, they are unlikely to harm the hosting web application directly, but they may cause problems for users.

Security Engineering Cheat Sheet

When Hosting XML-Based Document Formats

Assume that the payload may be interpreted as XHTML or some other script-enabled document type, regardless of the *Content-Type* and the top-level *xmlns* directive. Do not allow unconstrained attacker-controlled markup anywhere inside the file. Use the *Content-Disposition: attachment* if data is not meant to be viewed directly; <*img*> and feeds will still work.

On All Non-HTML Document Types

Use correct, browser-recognized *Content-Type* and *charset* values. Specify the *Content-Disposition: attachment* where possible. Verify and constrain output syntax. Consult the cheat sheet in Chapter 13 to avoid security problems related to content-sniffing flaws.

8

CONTENT RENDERING WITH BROWSER PLUG-INS

Browser plug-ins come in many forms and shapes, but the most common variety give the ability to display new file formats in the browser, as if they were HTML. The browser simply hands over the retrieved file, provides the helper application with a rectangular drawing surface in the document window, and essentially backs away from the scene. Such content-rendering plug-ins are clearly distinguished from browser extensions, a far more numerous bunch that commonly relies on JavaScript code to tweak how the already-supported, in-browser content is presented to the user.

Browser plug-ins have a long and colorful history of security flaws. In fact, according to some analysts, 12 out of the 15 most frequently exploited client-side vulnerabilities in 2010 could be attributed to the quality of plug-in software.[1] Many of these problems are because the underlying parsers were originally not meant to handle malicious inputs gracefully and have not benefited from the intense scrutiny that the remainder of the Web has been subject to. Other problems stem from the unusual security models devised by

plug-in developers and the interference between these permissions, the traditional design of web browsers, and the commonsense expectations of application developers.

We will review some of the security mechanisms used by popular plug-ins in the next chapter of this book. Before taking this dive, it makes sense to look at the ways plug-ins integrate with other online content and the common functionality they offer.

Invoking a Plug-in

Content-rendering plug-ins can be activated in a couple of ways. The most popular explicit method is to use *<embed src=...>* or *<object data=...>* markup in a "host" HTML document, with the *src* or *data* parameter pointing to the URL from which the actual plug-in-recognized document is to be retrieved. The dimensions and position of the drawable area allocated for the plug-in can be controlled with CSS (or with legacy HTML parameters).

In this scenario, every *<embed>* or *<object>* tag should be accompanied by an additional *type* parameter. The MIME type specified there will be compared to the list of MIME types registered by all the active plug-ins, and the retrieved file will be routed to the appropriate handler. If no match is found, a warning asking the user to download a plug-in should be theoretically displayed instead, although most browsers look at other signals before resorting to this unthinkable possibility; examining *Content-Type* or the apparent file extension spotted in the URL are two common choices.

NOTE *An obsolete* <applet> *tag, used to load Java programs (roughly equivalent to* <object type="application/x-java-applet">*), works in a comparable way but unconditionally disregards these auxiliary signals.*

Additional input to the plug-in is commonly passed using *<param>* tags nested inside the *<object>* block or through nonstandard additional parameters attached to the *<embed>* markup itself. The former, more modern approach may look like this:

```
<object data="app.swf" type="application/x-shockwave-flash">
  <param name="some_param1" value="some_value1">
  <param name="some_param2" value="some_value2">
  ...
</object>
```

In this content-inclusion mode, the *Content-Type* header returned by the server when retrieving the subresource is typically ignored, unless the *type* parameter is unknown to the browser. This is an unfortunate design, for reasons that will be explained shortly.

The other method for displaying plug-in content involves navigating directly to a suitable file. In this case, and in the case of *<embed>* or *<object>* with a missing *type* parameter, the *Content-Type* value obtained from the server is honored, and it will be compared with the list of plug-in-recognized MIME

types. If a match is found, the content is routed to the appropriate component. If the *Content-Type* lookup fails or the header is missing, some browsers will examine the response body for known content signatures; others just give up.

NOTE *The aforementioned content-focused methods aside, several types of plug-ins can be loaded directly from within JavaScript or VBScript programs without the need to explicitly create any HTML markup or retrieve any external data. Such is the case for ActiveX, an infamous script-to-system integration bridge available in Internet Explorer. (We will devote some time to ActiveX later in this chapter, but first things first.)*

The Perils of Plug-in Content-Type Handling

As noted in the previous section, in certain scenarios the *Content-Type* parameter on a retrieved plug-in-handled file is ignored, and the *type* parameter in the corresponding markup on the embedding page is used instead. While this decision is somewhat similar to the behavior of other type-specific content-inclusion tags (say, *<img>*), as discussed in "Type-Specific Content Inclusion" on page 82, it has some unique and ultimately disastrous consequences in the plug-in world.

The big problem is that several types of plug-ins are essentially full-fledged code execution environments and give the executed applications (*applets*) a range of special privileges to interact with the originating domain. For example, a Flash file retrieved from *fuzzybunnies.com* would be granted access to its originating domain (complete with a user's cookies) when embedded on the decidedly rogue *bunnyoutlet.com*.

In such a scenario, it would seem to be important for *fuzzybunnies.com* to be able to clearly communicate that a particular type of a document is indeed meant to be interpreted by a plug-in—and, consequently, that some documents aren't meant to be used this way. Unfortunately, there is no way for this to happen: The handling of a retrieved file is fully controlled by the embedding site (in our example, by the mean-spirited bullies who own *bunnyoutlet.com*). Therefore, if the originating domain hosts any type of user-controlled content, even in a nominally harmless format (such as *text/plain* or *image/jpeg*), the owners of *bunnyoutlet.com* may instruct the browser to disregard the existing metadata and route that document to a plug-in of their choice. A simple markup to achieve this sinister goal may be

```
<object data="http://fuzzybunnies.com/avatars/user11630.jpg"
        type="application/x-shockwave-flash">
```

If this turn of events seems wrong, that's because it is. Security researchers have repeatedly demonstrated that it is quite easy to construct documents that are, for example, simultaneously a valid image and a valid plug-in-recognized executable. The well-known "GIFAR" vulnerability, discovered in 2008 by Billy Rios,[2] exploited that very trick: It smuggled a Java applet inside a perfectly kosher GIF image. In response, Sun Microsystems reportedly tightened down the Java JAR file parser to mitigate the risk, but the general threat of such mistakes is still very real and will likely rear its ugly head once more.

Interestingly, the decision by some developers to rely on *Content-Type* and other signals if the *type* parameter is unrecognized is almost as bad. This decision makes it impossible for the well-intentioned *fuzzybunnies.com* to safely embed a harmless video from the rogues at *bunnyoutlet.com* by simply specifying *type="video/x-ms-wmv"*, because if any of the visitors do not have a plug-in for that specific media type, *bunnyoutlet.com* will suddenly have a say in what type of plug-in should be loaded on the embedding site instead. Some browsers, such as Internet Explorer, Chrome, or Opera, may also resort to looking for apparent file extensions present in the URL, which can lead to an interesting situation where neither the embedding nor the hosting party has real control over how a document is displayed—and quite often only the attacker is in charge.

A much safer design would require the embedder-controlled *type* parameter and the host-controlled *Content-Type* header to match (at least superficially). Unfortunately, there is currently no way to make this happen. Several individual plug-ins try to play nice (for example, following a 2008 overhaul, Adobe Flash rejects applets served with *Content-Disposition: attachment*, as does the built-in PDF reader in Chrome), but these improvements are few and far between.

Document Rendering Helpers

A significant portion of the plug-in landscape belongs to programs that allow certain very traditional, "nonweb" document formats to be shown directly in the browser. Some of these programs are genuinely useful: Windows Media Player, RealNetworks RealPlayer, and Apple QuickTime have been the backbone of online multimedia playback for about a decade, at least until their displacement by Adobe Flash. The merits of others are more questionable, however. For example, Adobe Reader and Microsoft Office both install in-browser document viewers, increasing the user's attack surface appreciably, though it is unclear whether these viewers offer a real benefit over opening the same document in a separate application with one extra click.

Of course, in a perfect world, hosting or embedding a PDF or a Word document should have no direct consequences for the security of the participating websites. Yet, predictably, the reality begs to differ. In 2009, a researcher noted that PDF-based forms that submit to *javascript:* URLs can apparently lead to client-side code execution on the embedding site.[3] Perhaps even more troubling than this report alone, according to that researcher's account, Adobe initially dismissed the report with the following note: "Our position is that, like an HTML page, a PDF file is active content."

It is regrettable that the hosting party does not have full control of when this active content is detected and executed and that otherwise reasonable webmasters may think of PDFs or Word documents as just a fancy way to present text. In reality, despite their harmless appearance, in a bid to look cool, many such document formats come equipped with their own hyperlinking capabilities or even scripting languages. For example, JavaScript code can be embedded in PDF documents, and Visual Basic macros are possible in

Microsoft Office files. When a script-bearing document is displayed on an HTML page, some form of a programmatic plug-in-to-browser bridge usually permits a degree of interaction with the embedding site, and the design of such bridges can vary from vaguely questionable to outright preposterous.

In one 2007 case, Petko D. Petkov noticed that a site that hosts any PDF documents can be attacked simply by providing completely arbitrary JavaScript code in the fragment identifier. This string will be executed on the hosting page through the plug-in bridge:[4]

```
http://example.com/random_document.pdf#foo=javascript:alert(1)
```

The two vulnerabilities outlined here are now fixed, but the lesson is that special care should be exercised when hosting or embedding any user-supplied documents in sensitive domains. The consequences of doing so are not well documented and can be difficult to predict.

Plug-in-Based Application Frameworks

The boring job of rendering documents is a well-established role for browser plug-ins, but several ambitious vendors go well beyond this paradigm. The aim of some plug-ins is simply to displace HTML and JavaScript by providing alternative, more featured platforms for building interactive web applications. That reasoning is not completely without merit: Browsers have long lacked in performance, in graphics capabilities, and in multimedia codecs, stifling some potential uses of the Web. Reliance on plug-ins is a reasonable short-term way to make a difference. On the flip side, when proprietary, patent- and copyright-encumbered plug-ins are promoted as the ultimate way to build an online ecosystem, without any intent to improve the browsers themselves, the openness of the Web inevitably suffers. Some critics, notably Steve Jobs, think that creating a tightly controlled ecosystem is exactly what several plug-in vendors, most notably Adobe, aspire to.[5]

In response to this perceived threat of a hostile takeover of the Web, many of the shortcomings that led to the proliferation of alternative application frameworks are now being hastily addressed under the vaguely defined umbrella of HTML5; *<video>* tags and WebGL* are the prime examples of this work. That said, some of the features available in plug-ins will probably not be captured as a part of any browser standard in the immediate future. For example, there is currently no serious plan to add inherently dangerous elevated privilege programs supported by Java or security-by-obscurity content protection schemes (euphemistically called Digital Rights Management, or DRM).

Therefore, while the landscape will change dramatically in the coming years, we can expect that in one form or another, proprietary web application frameworks are here to stay.

* WebGL is a fairly recent attempt to bring OpenGL-based 3D graphics to JavaScript applications. The first specification of the standard appeared in March 2011, and wide browser-level support is expected to follow.

Adobe Flash

Adobe Flash is a web application framework introduced in 1996, in the heat of the First Browser Wars. Before its acquisition by Adobe in 2005, the Flash platform was known as Macromedia Flash or Shockwave Flash (hence the *.swf* file extension used for Flash files), and it is still sometimes referred to as such.

Flash is a fairly down-to-earth platform built on top of a JavaScript-based language dubbed ActionScript.[7] It includes a 2-D vector and bitmap graphics-rendering engine and built-in support for several image, video, and audio formats, such as the popular and efficient H.264 codec (which is used for much of today's online multimedia).

By most estimates, Flash is installed on around 95 to 99 percent of all desktop systems.[8, 9] This user base is substantially higher than that of any other media player plug-in. (Support for the Windows Media Player and QuickTime plug-ins is available on only about 60 percent of PCs, despite aggressive bundling strategies, while the increasingly unpopular RealPlayer is still clinging to 25 percent.) The market position contributes to the product's most significant and unexpected use: the replacement of all multimedia playback plug-ins previously relied upon for streaming video on the Web. Although the plug-in is also used for a variety of other jobs (including implementing online games, interactive advertisements, and so on), simple multimedia constitutes a disproportionately large slice of the pie.

NOTE *Confusingly, a separate plug-in called Adobe Shockwave Player (without the word "Flash") is also available, which can be used to play back content created with Adobe Director. This plug-in is sometimes mistakenly installed in place of or alongside Adobe Flash, contributing to an approximately 20 percent install base,[6] but it is almost always unnecessary. The security properties of this plug-in are not particularly well studied.*

Properties of ActionScript

The capabilities of ActionScript in SWF files are generally analogous to those of JavaScript code embedded on HTML pages with some minor, yet interesting, differences. For example, Flash programs are free to enumerate all fonts installed on a system and collect other useful system fingerprinting signals not available to normal scripts. Flash programs can also use full screen rendering, facilitating UI spoofing attacks, and they can request access to input devices such as a camera or a microphone (this requires the user's consent). Flash also tends to ignore browser security and privacy settings and uses its own configuration for mechanisms such as in-plug-in persistent data storage (although some improvements in this area were announced in May 2011).

The remaining features are less surprising. We'll discuss the network and DOM access permissions of Flash applications in more detail in the next chapter, but in short, by default, every Flash applet can use the browser HTTP stack (and any ambient credentials managed therein) to talk back to its originating server, request a limited range of subresources from other sites, and navigate the current browser window or open a new one. ActionScript programs may also negotiate browser-level access to other currently running

Flash applications and, in some cases, access the DOM of the embedding page. This last functionality is implemented by injecting *eval(...)*-like statements into the target JavaScript context.

ActionScript offers fertile ground for web application vulnerabilities. For example, the *getURL(...)* and *navigateToURL(...)* functions, used to navigate the browser or open new windows, are sometimes invoked with attacker-controlled inputs. Such a use is dangerous. Even though *javascript:* URLs do not have a special meaning to Flash, the function will pass such strings to the browser, in some cases resulting in script injection on the embedding site.

Until recently, a related problem was present with other URL-handling APIs, such as *loadMovie(...)*. Even though the function did not rely on the browser to load the document, it would recognize an internal *asfunction:* scheme, which works similarly to *eval(...)* and could be trivially leveraged to perform a call to *getURL(...)*:

```
asfunction:getURL,javascript:alert('Hi mom!')
```

The issue with loading scripts from untrusted sources, discussed in Chapter 6, also has an equivalent in the plug-in word. In Flash, it is very unsafe to invoke certain functions that affect the state of the ActionScript execution environment (such as the *LoadVars.load(...)*) with attacker-controlled URLs, even if the scheme from which the resource is loaded is *http:* or *https:*.

Another commonly overlooked attack surface is the internal, simplified HTML parser offered by the Flash plug-in: Basic HTML markup can be assigned to properties such as *TextField.htmlText* and *TextArea.htmlText*. It is easy to forget that user-supplied content must be escaped correctly in this setting. Failure to do so may permit attackers to modify the appearance of the application UI or to inject potentially problematic scripting-oriented links.

Yet another class of Flash-related security bugs may arise due to design or implementation problems in the plug-in itself. For example, take the *ExternalInterface.call(...)* API. It is meant to allow ActionScript to call existing JavaScript functions on the embedding page and takes two parameters: the name of the JavaScript function to call and an optional string to be passed to this routine. While it is understood that the first parameter should not be attacker controlled, it appears to be safe to put user data in the second one. In fact, the documentation provides the following code snippet outlining this specific use case:[10]

```
ExternalInterface.call("sendToJavaScript", input.text);
```

This call will result in the following *eval(...)* statement being injected on the embedding page:

```
try {
  __flash__toXML(sendToJavaScript, "value of input.text"));
} catch (e) {
  "<undefined/>";
}
```

When writing the code behind this call, the authors of the plug-in remembered to use backslash escaping when outputting the second parameter: *hello"world* becomes *hello\"world*. Unfortunately, they overlooked the need to escape any stray backslash characters, too. Because of this, if the value of *input.text* is set to the following string, the embedded script will unexpectedly execute:

```
Hello world!\"+alert(1); } catch(e) {} //
```

I contacted Adobe about this particular problem in March 2010. Over a year later, its response was this: "We have not made any change to this behavior for backwards compatibility reasons."

That seems unfortunate.

Microsoft Silverlight

Microsoft Silverlight is a versatile development platform built on the Windows Presentation Foundation, a GUI framework that is a part of Microsoft's .NET stack. It debuted in 2007 and combines an Extensible Application Markup Language (XAML)[11] (Microsoft's alternative to Mozilla's XUL) with code written in one of several managed .NET languages,* such as C# or Visual Basic.

Despite substantial design differences and a more ambitious (and confusing) architecture, this plug-in is primarily meant to compete with Adobe Flash. Many of the features available to Silverlight applications mirror those implemented in its competitor, including a nearly identical security model and a similar *eval(...)*-based bridge to the embedding page. To Microsoft's credit, Silverlight does not come with an equivalent of the *asfunction:* scheme or with a built-in HTML renderer, however.

Silverlight is marketed by Microsoft fairly aggressively, and it is bundled with some editions of Internet Explorer. As a result, depending on the source, it is believed to have about a 60 to 75 percent desktop penetration.[12] Despite its prevalence, Silverlight is used fairly infrequently to develop actual web applications, perhaps because it usually offers no compelling advantages over its more established counterpart or because its architecture is seen as more contrived and platform-specific. (Netflix, a popular video streaming and rental service, is one of the very few high-profile websites that actually relies on Silverlight for playback on some devices.)

Sun Java

Java is a programming language coupled with a platform-independent, managed-code execution platform. Developed in the early to mid-1990s by James Gosling for Sun Microsystems, Java has a well-established role as a server-side programming language and a very robust presence in many other niches,

* Managed code is not executed directly by the CPU (which would be inherently unsafe, because CPUs are not designed to enforce web security rules). Rather, it is compiled to an intermediate binary form and then interpreted at runtime by a specialized virtual machine. This approach is faster than interpreting scripts at runtime and permits custom security policy enforcement as the program is being executed.

including mobile devices. Yet, from the beginning, Sun hoped that Java would also occupy a prominent place on the browser end.

Java in the browser predated Flash and most similar plug-ins, and the now-obsolete *<applet>* tag is a testament to how important and unique and novel this addition must have seemed back in its day. Yet, despite this head start, the Java language is nearly extinct as an in-browser development platform, and even in its heyday it never enjoyed real prominence. It retains a remarkable 80 percent installed base, but this high percentage is attributed largely to the fact that the Java plug-in is bundled with Java Runtime Environment (JRE), a more practically useful and commonly preinstalled component that is required to run normal, desktop Java applications on the system without any involvement on the browser end.

The reasons for the failure of Java as a browser technology are difficult to pinpoint. Perhaps it's due to the plug-in's poor startup performance, the clunky UI libraries that made it difficult to develop snappy and user-friendly web applications, or the history of vicious litigation between Sun and Microsoft that cast a long shadow over the future of the language on Microsoft's operating systems.* Whatever the reasons may be, the high install base of Java coupled with its marginal use means that the risks it creates far outweigh any potential benefits to the users. (The plug-in had close to 80 security vulnerabilities in 2010,[13] and the vendor is commonly criticized for patching such bugs very slowly.)

Java's security policies are somewhat similar to those of other plug-ins, but in some aspects, such as its understanding of the same-origin policy or its ability to restrict access to the embedding page, it compares unfavorably. (The next chapter provides an overview of this.) It is also worth noting that unlike with Flash or Silverlight, certain types of cryptographically signed applets may request access to potentially dangerous OS features, such as unconstrained networking or file access, and only a user's easily coaxed consent stands in the way.

XML Browser Applications (XBAP)

XML Browser Applications (XBAP)[14] is Microsoft's heavy-handed foray into the world of web application frameworks, attempted in the years during which the battle over Java started going sour and before the company released Silverlight.

XBAP is reminiscent of Silverlight in that it leverages the same Windows Presentation Foundation and .NET architecture. However, instead of being a self-contained and snappy browser plug-in, it depends on the large and unwieldy .NET runtime, in a manner similar to the Java plug-in's dependence on JRE. It executes the managed code in a separate process called *PresentationHost.exe*, often loading extensive dependencies at initialization time. By Microsoft's own admission, the load time of a medium-size previously uncached application

* The legal battles started in 1997, when Microsoft decided to roll out its own (and in some ways, superior) version of the Java virtual machine. Sun Microsystems sued, hoping to win an injunction that would force Microsoft to bundle Sun's version instead. The two companies initially settled in 2001, but shortly thereafter they headed back to court. In the final settlement in 2004, Sun walked away with $1.6 billion in cash, but Windows users were not getting any Java runtime at all.

could easily reach 10 seconds or more. When the technology premiered in 2002, most users were already expecting Internet applications to be far more responsive than that.

The security model of XBAP applications is poorly documented and has not been researched to date, perhaps due to XBAP's negligible real-world use and obtuse, multilayer architecture. One would reasonably expect that XBAP's security properties would parallel the model eventually embraced for Silverlight, but with broader access to certain .NET libraries and UI widgets. And, apparently as a result of copying from Sun, XBAP programs can also be given elevated privileges when loaded from the local filesystem or signed with a cryptographic certificate.

Microsoft bundled XBAP plug-ins with its .NET framework to the point of silently installing nonremovable Windows Presentation Foundation plug-ins—not only in Internet Explorer but also in the competing Firefox and Chrome. This move stirred some well-deserved controversy, especially once the first vulnerability reports started pouring in. (Mozilla even temporarily disabled the plug-in through an automated update to protect its users.) Still, despite such bold and questionable moves to popularize it, nobody actually wanted to write XBAP applets, and inch by inch, the technology followed Java into the dustbin of history.

Eventually, Microsoft appeared to acknowledge this failure and chose to focus on Silverlight instead. Beginning with Internet Explorer 9, XBAP is disabled by default for Internet-originating content, and the dubious Firefox and Chrome plug-ins are no longer automatically pushed to users. Nevertheless, it seems reasonable to assume that at least 10 percent of all Internet users may be still browsing with a complex, partly abandoned, and largely unnecessary plug-in installed on their machines and will continue to do so for the next couple of years.

ActiveX Controls

At its core, ActiveX is the successor to Object Linking and Embedding (OLE), a 1990 technology that made it possible for programs to reuse components of other applications in a standardized, language-independent way. A simple use case for ActiveX would be a spreadsheet application wishing to embed an editable vector image from a graphics-editing program or a simple game that wants to embed a video player.

The idea is not controversial, but by the mid-1990s Microsoft had decided that ActiveX made sense in the browser, too. After all, wouldn't websites want to benefit from the same Windows components that desktop applications could rely on? The approach violates the idea of nurturing an open, OS-independent web, but it's otherwise impressive, as illustrated by the following JavaScript example that casually creates, edits, and saves an Excel spreadsheet:

```
var sheet = new ActiveXObject("Excel.Sheet");
sheet.ActiveSheet.Cells(42,42).Value = "Hi mom!";
sheet.SaveAs("c:\\spreadsheet.xls");
sheet.Application.Quit();
```

Standards compliance aside, Microsoft's move to ActiveX proved disastrous from a security standpoint. Many of the exposed ActiveX components were completely unprepared to behave properly when interacting with untrusted environments, and over the next 15 years, researchers discovered several hundred significant security vulnerabilities in web-accessible ActiveX controls. Heck, the simple observation that Firefox does not support this technology helped bolster its security image at the onset of the Second Browser Wars.

Despite this fiasco, Microsoft stood by ActiveX defiantly, investing in gradually limiting the number of controls that could be accessed from the Internet and fixing the bugs in those it considered essential. Not until Internet Explorer 9 did Microsoft finally decide to let go: Internet Explorer 9 disables all ActiveX access by default, requiring several extra clicks to use it when needed.

NOTE *The wisdom of delegating the choice to the user is unclear, especially since the permission granted to a site extends not only to legitimate content on that website but also to any payloads injected due to application bugs such as XSS. Still, Internet Explorer 9 is some improvement.*

Living with Other Plug-ins

So far, we have covered almost all general-purpose browser plug-ins in use today. Although there is a long tail of specialized or experimental plug-ins, their use is fairly insignificant and not something that we need to take into account when surveying the overall health of the online ecosystem.

Well, with one exception. An unspecified but probably significant percentage of online users can be expected to have an assortment of web-exposed browser plug-ins or ActiveX controls that they never knowingly installed, or that they were forced to install even though it's doubtful that they would ever benefit from the introduced functionality.

This inexcusable practice is sometimes embraced by otherwise reputable and trusted companies. For example, Adobe forces users who wish to download Adobe Flash to also install GetRight, a completely unnecessary third-party download utility. Microsoft does the same with Akamai Download Manager on its developer-oriented website, complete with a hilarious justification (emphasis mine):[15]

> What is the Akamai Download Manager and why do I *have* to use it?
>
> ***To help you*** download large files with reduced chance of interruption, ***some downloads require*** the use of the Akamai Download Manager.

The primary concern with software installed this way and exposed directly to malicious input from anywhere on the Internet is that unless it is designed with extreme care, it is likely to have vulnerabilities (and sure enough, both GetRight and Akamai Download Manager had some). Therefore, the risks of browsing with a completely unnecessary plug-in that only served a particular purpose once or twice far outweigh the purported (and usually unwanted) benefits.

Security Engineering Cheat Sheet

When Serving Plug-in-Handled Files

- ☑ **Data from trusted sources:** Data from trusted sources is generally safe to host, but remember that security vulnerabilities in Flash, Java, or Silverlight applets, or in the Adobe Reader JavaScript engine, may impact the security of your domain. Avoid processing user-supplied URLs and generating or modifying user-controlled HTML from within plug-in-executed applets. Exercise caution when using the JavaScript bridge.
- ☑ **User-controlled simple multimedia:** User-controlled multimedia is relatively safe to host, but be sure to validate and constrain the format, use the correct *Content-Type*, and consult the cheat sheet in Chapter 13 to avoid security problems caused by content-sniffing flaws.
- ☑ **User-controlled document formats:** These are not inherently unsafe, but they have an increased risk of contributing security problems due to plug-in design flaws. Consider hosting from a dedicated domain when possible. If you need to authenticate the request to an isolated domain, do so with a single-use request token instead of by relying on cookies.
- ☑ **User-controlled active applications:** These are unsafe to host in sensitive domains.

When Embedding Plug-in-Handled Files

Always make sure that plug-in content on HTTPS sites is also loaded over HTTPS,[*] and always explicitly specify the *type* parameter on *<object>* or *<embed>*. Note that because of the non-authoritative handling of *type* parameters, restraint must be exercised when embedding plug-in content from untrusted sources, especially on highly sensitive sites.

- ☑ **Simple multimedia:** It is generally safe to load simple multimedia from third-party sources, with the caveats outlined above.
- ☑ **Document formats:** These are usually safe, but they carry a greater potential for plug-in and browser content-handling issues than simple multimedia. Exercise caution.
- ☑ **Flash and Silverlight:** In principle, Flash and Silverlight apps can be embedded safely from external sources if the appropriate security flags are present in the markup. If the flags are not specified correctly, you may end up tying the security of your site to that of the provider of the content. Consult the cheat sheet in Chapter 9 for advice.
- ☑ **Java:** Java always ties the security of your service to that of the provider of the content, because DOM access to the embedding page can't be reliably restricted. See Chapter 9. Do not load Java apps from untrusted sites.

If You Want to Write a New Browser Plug-in or ActiveX Component

Unless you are addressing an important, common-use case that will benefit a significant fraction of the Internet, please reconsider. If you are scratching an important itch, consider doing it in a peer-reviewed, standardized manner as a part of HTML5.

[*] If loading an HTTP-delivered applet on an HTTPS page is absolutely unavoidable, it is safer to place it inside an intermediate HTTP frame rather than directly inside the HTTPS document, as this prevents the applet-to-JavaScript bridge from being leveraged for attacks.

PART II

BROWSER SECURITY FEATURES

Having reviewed the basic building blocks of the Web, we can now comfortably examine all the security features that keep rogue web applications at bay. Part II of this book takes a look at everything from the well-known but often misunderstood same-origin policy to the obscure and proprietary zone settings of Internet Explorer. It explains what these mechanisms can do for you—and when they tend to fall apart.

9

CONTENT ISOLATION LOGIC

Most of the security assurances provided by web browsers are meant to isolate documents based on their origin. The premise is simple: Two pages from different sources should not be allowed to interfere with each other. Actual practice can be more complicated, however, as no universal agreement exists about where a single document begins and ends or what constitutes a single origin. The result is a sometimes unpredictable patchwork of contradictory policies that don't quite work well together but that can't be tweaked without profoundly affecting all current legitimate uses of the Web.

These problems aside, there is also little clarity about what actions should be subject to security checks in the first place. It seems clear that some interactions, such as following a link, should be permitted without special restrictions as they are essential to the health of the entire ecosystem, and that others, such as modifying the contents of a page loaded in a separate window, should require a security check. But a large gray area exists between these extremes, and that middle ground often feels as if it's governed more by a roll of the dice than by any unified plan. In these murky waters, vulnerabilities such as cross-site request forgery (see Chapter 4) abound.

It's time to start exploring. Let's roll a die of our own and kick off the journey with JavaScript.

Same-Origin Policy for the Document Object Model

The *same-origin policy* (*SOP*) is a concept introduced by Netscape in 1995 alongside JavaScript and the Document Object Model (DOM), just one year after the creation of HTTP cookies. The basic rule behind this policy is straightforward: Given any two separate JavaScript execution contexts, one should be able to access the DOM of the other only if the protocols, DNS names,* and port numbers associated with their host documents match exactly. All other cross-document JavaScript DOM access should fail.

The protocol-host-port tuple introduced by this algorithm is commonly referred to as *origin*. As a basis for a security policy, this is pretty robust: SOP is implemented across all modern browsers with a good degree of consistency and with only occasional bugs.† In fact, only Internet Explorer stands out, as it ignores the port number for the purpose of origin checks. This practice is somewhat less secure, particularly given the risk of having non-HTTP services running on a remote host for HTTP/0.9 web servers (see Chapter 3). But usually it makes no appreciable difference.

Table 9-1 illustrates the outcome of SOP checks in a variety of situations.

Table 9-1: Outcomes of SOP Checks

Originating document	Accessed document	Non–IE browser	Internet Explorer
http://example.com/**a/**	http://example.com/**b/**	Access okay	Access okay
http://example.com/	http://**www.**example.com/	Host mismatch	Host mismatch
http://example.com/	**https**://example.com/	Protocol mismatch	Protocol mismatch
http://example.com**:81**/	http://example.com/	Port mismatch	Access okay

NOTE *This same-origin policy was originally meant to govern access only to the DOM ; that is, the methods and properties related to the contents of the actual displayed document. The policy has been gradually extended to protect other obviously sensitive areas of the root JavaScript object, but it is not all-inclusive. For example, non-same-origin scripts can usually still call* location.assign() *or* location.replace(...) *on an arbitrary window or a frame. The extent and the consequences of these exemptions are the subject of Chapter 11.*

* This and most other browser security mechanisms are based on DNS labels, not on examining the underlying IP addresses. This has a curious consequence: If the IP of a particular host changes, the attacker may be able to talk to the new destination through the user's browser, possibly engaging in abusive behaviors while hiding the true origin of the attack (unfortunate, not very interesting) or interacting with the victim's internal network, which normally would not be accessible due to the presence of a firewall (a much more problematic case). Intentional change of an IP for this purpose is known as *DNS rebinding*. Browsers try to mitigate DNS rebinding to some extent by, for example, caching DNS lookup results for a certain time (*DNS pinning*), but these defenses are imperfect.

† One significant source of same-origin policy bugs is having several separate URL-parsing routines in the browser code. If the parsing approach used in the HTTP stack differs from that used for determining JavaScript origins, problems may arise. Safari, in particular, combated a significant number of SOP bypass flaws caused by pathological URLs, including many of the inputs discussed in Chapter 2.

The simplicity of SOP is both a blessing and a curse. The mechanism is fairly easy to understand and not too hard to implement correctly, but its inflexibility can be a burden to web developers. In some contexts, the policy is too broad, making it impossible to, say, isolate home pages belonging to separate users (short of giving each a separate domain). In other cases, the opposite is true: The policy makes it difficult for legitimately cooperating sites (say, *login.example.com* and *payments.example.com*) to seamlessly exchange data.

Attempts to fix the first problem—to narrow down the concept of an origin—are usually bound to fail because of interactions with other explicit and hidden security controls in the browser. Attempts to broaden origins or facilitate cross-domain interactions are more common. The two broadly supported ways of achieving these goals are *document.domain* and *postMessage(…)*, as discussed below.

document.domain

This JavaScript property permits any two cooperating websites that share a common top-level domain (such as *example.com*, or even just *.com*) to agree that for the purpose of future same-origin checks, they want to be considered equivalent. For example, both *login.example.com* and *payments.example.com* may perform the following assignment:

```
document.domain = "example.com"
```

Setting this property overrides the usual hostname matching logic during same-origin policy checks. The protocols and port numbers still have to match, though; if they don't, tweaking *document.domain* will not have the desired effect.

Both parties must explicitly opt in for this feature. Simply because *login.example.com* has set its *document.domain* to *example.com* does not mean that it will be allowed to access content originating from the website hosted at *http://example.com/*. That website needs to perform such an assignment, too, even if common sense would indicate that it is a no-op. This effect is symmetrical. Just as a page that sets *document.domain* will not be able to access pages that did not, the action of setting the property also renders the caller mostly (but not fully!)[*] out of reach of normal documents that previously would have been considered same-origin with it. Table 9-2 shows the effects of various values of *document.domain*.

Despite displaying a degree of complexity that hints at some special sort of cleverness, *document.domain* is not particularly safe. Its most significant weakness is that it invites unwelcome guests. After two parties mutually set this property to *example.com*, it is not simply the case that *login.example.com* and *payments.example.com* will be able to communicate; *funny-cat-videos.example.com* will be able to jump on the bandwagon as well. And because of the degree

[*] For example, in Internet Explorer, it will still be possible for one page to navigate any other documents that were nominally same-origin but that became "isolated" after setting *document.domain*, to *javascript:* URLs. Doing so permits any JavaScript to execute in the context of such as a pseudo-isolated domain. On top of this, obviously nothing stops the originating page from simply setting its own *document.domain* to a value identical with that of the target in order to eliminate the boundary. In other words, the ability to make a document non-same-origin with other pages through *document.domain* should not be relied upon for anything even remotely serious or security relevant.

of access permitted between the pages, the integrity of any of the participating JavaScript contexts simply cannot be guaranteed to any realistic extent. In other words, touching *document.domain* inevitably entails tying the security of your page to the security of the weakest link in the entire domain. An extreme case of setting the value to **.com* is essentially equivalent to assisted suicide.

Table 9-2: Outcomes of *document.domain* Checks

Originating document		Accessed document		Outcome
URL	document.domain	URL	document.domain	
http://www.example.com/	*example.com*	*http://payments.example.com/*	*example.com*	Access okay
http://www.example.com/	*example.com*	*https://payments.example.com/*	*example.com*	Protocol mismatch
http://payments.example.com/	*example.com*	*http://example.com/*	**(not set)**	Access denied
http://www.example.com/	**(not set)**	*http://www.example.com/*	*example.com*	Access denied

postMessage(...)

The *postMessage(...)* API is an HTML5 extension that permits slightly less convenient but remarkably more secure communications between non-same-origin sites without automatically giving up the integrity of any of the parties involved. Today it is supported in all up-to-date browsers, although because it is fairly new, it is not found in Internet Explorer 6 or 7.

The mechanism permits a text message of any length to be sent to any window for which the sender holds a valid JavaScript handle (see Chapter 6). Although the same-origin policy has a number of gaps that permit similar functionality to be implemented by other means,[*] this one is actually safe to use. It allows the sender to specify what origins are permitted to receive the message in the first place (in case the URL of the target window has changed), and it provides the recipient with the identity of the sender so that the integrity of the channel can be ascertained easily. In contrast, legacy methods that rely on SOP loopholes usually don't come with such assurances; if a particular action is permitted without robust security checks, it can usually also be triggered by a rogue third party and not just by the intended participants.

To illustrate the proper use of *postMessage(...)*, consider a case in which a top-level document located at *payments.example.com* needs to obtain user login information for display purposes. To accomplish this, it loads a frame pointing to *login.example.com*. This frame can simply issue the following command:

```
parent.postMessage("user=bob", "https://payments.example.com");
```

[*] More about this in Chapter 11, but the most notable example is that of encoding data in URL fragment identifiers. This is possible because navigating frames to a new URL is not subject to security restrictions in most cases, and navigation to a URL where only the fragment identifier changes does not actually trigger a page reload. Framed JavaScipt can simply poll *location.hash* and detect incoming messages this way.

The browser will deliver the message only if the embedding site indeed matches the specified, trusted origin. In order to securely process this response, the top-level document needs to use the following code:

```
// Register the intent to process incoming messages:
addEventListener("message", user_info, false);

// Handle actual data when it arrives:
function user_info(msg) {
  if (msg.origin == "https://login.example.com") {
    // Use msg.data as planned
  }
}
```

PostMessage(...) is a very robust mechanism that offers significant benefits over *document.domain* and over virtually all other guerrilla approaches that predate it; therefore, it should be used as often as possible. That said, it can still be misused. Consider the following check that looks for a substring in the domain name:

```
if (msg.origin.indexOf(".example.com") != -1) { ... }
```

As should be evident, this comparison will not only match sites within *example.com* but will also happily accept messages from *www.example.com.bunnyoutlet.com*. In all likelihood, you will stumble upon code like this more than once in your journeys. Such is life!

NOTE *Recent tweaks to HTML5 extended the* postMessage(...) *API to incorporate somewhat overengineered "ports" and "channels," which are meant to facilitate stream-oriented communications between websites. Browser support for these features is currently very limited and their practical utility is unclear, but from the security standpoint, they do not appear to be of any special concern.*

Interactions with Browser Credentials

As we are wrapping up the overview of the DOM-based same-origin policy, it is important to note that it is in no way synchronized with ambient credentials, SSL state, network context, or many other potentially security-relevant parameters tracked by the browser. Any two windows or frames opened in a browser will remain same-origin with each other even if the user logs out from one account and logs into another, if the page switches from using a good HTTPS certificate to a bad one, and so on.

This lack of synchronization can contribute to the exploitability of other security bugs. For example, several sites do not protect their login forms against cross-site request forgery, permitting any third-party site to simply submit a username and a password and log the user into an attacker-controlled account. This may seem harmless at first, but when the content loaded in the browser before and after this operation is considered same-origin, the impact of normally ignored "self-inflicted" cross-site scripting vulnerabilities (i.e., ones where the owner of a particular account can target only himself) is suddenly

much greater than it would previously appear. In the most basic scenario, the attacker may first open and keep a frame pointing to a sensitive page on the targeted site (e.g., *http://www.fuzzybunnies.com/address_book.php*) and then log the victim into the attacker-controlled account to execute self-XSS in an unrelated component of *fuzzybunnies.com*. Despite the change of HTTP credentials, the code injected in that latter step will have unconstrained access to the previously loaded frame, permitting data theft.

Same-Origin Policy for XMLHttpRequest

The *XMLHttpRequest* API, mentioned in this book on several prior occasions, gives JavaScript programs the ability to issue almost unconstrained HTTP requests to the server from which the host document originated, and read back response headers and the document body. The ability to do so would not be particularly significant were it not for the fact that the mechanism leverages the existing browser HTTP stack and its amenities, including ambient credentials, caching mechanisms, keep-alive sessions, and so on.

A simple and fairly self-explanatory use of a synchronous *XMLHttpRequest* could be as follows:

```
var x = new XMLHttpRequest();
x.open("POST", "/some_script.cgi", false);
x.setRequestHeader("X-Random-Header", "Hi mom!");
x.send("...POST payload here...");
alert(x.responseText);
```

Asynchronous requests are very similar but are executed without blocking the JavaScript engine or the browser. The request is issued in the background, and an event handler is called upon completion instead.

As originally envisioned, the ability to issue HTTP requests via this API and to read back the data is governed by a near-verbatim copy of the same-origin policy with two minor and seemingly random tweaks. First, the *document.domain* setting has no effect on this mechanism, and the destination URL specified for *XMLHttpRequest.open(...)* must always match the true origin of the document. Second, in this context, port number is taken into account in Internet Explorer versions prior to 9, even though this browser ignores it elsewhere.

The fact that *XMLHttpRequest* gives the user an unprecedented level of control over the HTTP headers in a request can actually be advantageous to security. For example, inserting a custom HTTP header, such as *X-Coming-From: same-origin*, is a very simple way to verify that a particular request is not coming from a third-party domain, because no other site should be able to insert a custom header into a browser-issued request. This assurance is not very strong, because no specification says that the implicit restriction on cross-domain headers can't change;[*] nevertheless, when it comes to web security, such assumptions are often just something you have to learn to live with.

Control over the structure of an HTTP request can also be a burden, though, because inserting certain types of headers may change the meaning of a request to the destination server, or to the proxies, without the browser

realizing it. For example, specifying an incorrect *Content-Length* value may allow an attacker to smuggle a second request into a keep-alive HTTP session maintained by the browser, as shown here.

```
var x = new XMLHttpRequest();
x.open("POST", "http://www.example.com/", false);

// This overrides the browser-computed Content-Length header:
x.setRequestHeader("Content-Length", "7");

// The server will assume that this payload ends after the first
// seven characters, and that the remaining part is a separate
// HTTP request.
x.send(
  "Gotcha!\n" +
  "GET /evil_response.html HTTP/1.1\n" +
  "Host: www.bunnyoutlet.com\n\n"
);
```

If this happens, the response to that second, injected request may be misinterpreted by the browser later, possibly poisoning the cache or injecting content into another website. This problem is especially pronounced if an HTTP proxy is in use and all HTTP requests are sent through a shared channel.

Because of this risk, and following a lot of trial and error, modern browsers blacklist a selection of HTTP headers and request methods. This is done with relatively little consistency: While *Referer*, *Content-Length*, and *Host* are universally banned, the handling of headers such as *User-Agent*, *Cookie*, *Origin*, or *If-Modified-Since* varies from one browser to another. Similarly, the TRACE method is blocked everywhere, because of the unanticipated risk it posed to *httponly* cookies—but the CONNECT method is permitted in Firefox, despite carrying a vague risk of messing with HTTP proxies.

Naturally, implementing these blacklists has proven to be an entertaining exercise on its own. Strictly for your amusement, consider the following cases that worked in some browsers as little as three years ago:[1]

```
XMLHttpRequest.setRequestHeader("X-Harmless", "1\nOwned: Gotcha");
```

or

```
XMLHttpRequest.setRequestHeader("Content-Length: 123 ", "");
```

or simply

```
XMLHttpRequest.open("GET\thttp://evil.com\tHTTP/1.0\n\n", "/", false);
```

[*] In fact, many plug-ins had problems in this area in the past. Most notably, Adobe Flash permitted arbitrary cross-domain HTTP headers until 2008, at which point its security model underwent a substantial overhaul. Until 2011, the same plug-in suffered from a long-lived implementation bug that caused it to resend any custom headers to an unrelated server following an attacker-supplied HTTP 307 redirect code. Both of these problems are fixed now, but discovery-to-patch time proved troubling.

NOTE Cross-Origin Resource Sharing[2] (CORS) *is a proposed extension to* XMLHttpRequest *that permits HTTP requests to be issued across domains and then read back if a particular response header appears in the returned data. The mechanism changes the semantics of the API discussed in this session by allowing certain "vanilla" cross-domain requests, meant to be no different from regular navigation, to be issued via* XMLHttpRequest.open(...) *with no additional checks; more elaborate requests require an OPTIONS-based preflight request first. CORS is already available in some browsers, but it is opposed by Microsoft engineers, who pursued a competing* XDomainRequest *approach in Internet Explorer 8 and 9. Because the outcome of this conflict is unclear, a detailed discussion of CORS is reserved for Chapter 16, which provides a more systematic overview of upcoming and experimental mechanisms.*

Same-Origin Policy for Web Storage

Web storage is a simple database solution first implemented by Mozilla engineers in Firefox 1.5 and eventually embraced by the HTML5 specification.[3] It is available in all current browsers but not in Internet Explorer 6 or 7.

Following several dubious iterations, the current design relies on two simple JavaScript objects: *localStorage* and *sessionStorage*. Both objects offer an identical, simple API for creating, retrieving, and deleting name-value pairs in a browser-managed database. For example:

```
localStorage.setItem("message", "Hi mom!");
alert(localStorage.getItem("message"));
localstorage.removeItem("message");
```

The *localStorage* object implements a persistent, origin-specific storage that survives browser shutdowns, while *sessionStorage* is expected to be bound to the current browser window and provide a temporary caching mechanism that is destroyed at the end of a browsing session. While the specification says that both *localStorage* and *sessionStorage* should be associated with an SOP-like origin (the protocol-host-port tuple), implementations in some browsers do not follow this advice, introducing potential security bugs. Most notably, in Internet Explorer 8, the protocol is not taken into account when computing the origin, putting HTTP and HTTPS pages within a shared context. This design makes it very unsafe for HTTPS sites to store or read back sensitive data through this API. (This problem is corrected in Internet Explorer 9, but there appears to be no plan to backport the fix.)

In Firefox, on the other hand, the *localStorage* behaves correctly, but the *sessionStorage* interface does not. HTTP and HTTPS use a shared storage context, and although a check is implemented to prevent HTTP content from reading keys created by HTTPS scripts, there is a serious loophole: Any key first created over HTTP, and then updated over HTTPS, will remain visible to nonencrypted pages. This bug, originally reported in 2009,[4] will eventually be resolved, but when is not clear.

Security Policy for Cookies

We discussed the semantics of HTTP cookies in Chapter 3, but that discussion left out one important detail: the security rules that must be implemented to protect cookies belonging to one site from being tampered with by unrelated pages. This topic is particularly interesting because the approach taken here predates the same-origin policy and interacts with it in a number of unexpected ways.

Cookies are meant to be scoped to domains, and they can't be limited easily to just a single hostname value. The *domain* parameter provided with a cookie may simply match the current hostname (such as *foo.example.com*), but this will not prevent the cookie from being sent to any eventual subdomains, such as *bar.foo.example.com*. A qualified right-hand fragment of the hostname, such as *example.com*, can be specified to request a broader scope, however.

Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the *domain* parameter, but this method will have no effect in Internet Explorer.

Table 9-3 illustrates cookie-setting behavior in some distinctive cases.

Table 9-3: A Sample of Cookie-Setting Behaviors

Cookie set at *foo.example.com*, *domain* parameter is:	**Scope of the resulting cookie**	
	Non–IE browsers	**Internet Explorer**
(value omitted)	*foo.example.com* (exact)	**.foo.example.com*
bar.foo.example.com	Cookie not set: domain more specific than origin	
foo.example.com	**.foo.example.com*	
baz.example.com	Cookie not set: domain mismatch	
example.com	**.example.com*	
ample.com	Cookie not set: domain mismatch	
.com	Cookie not set: domain too broad, security risk	

The only other true cookie-scoping parameter is the path prefix: Any cookie can be set with a specified *path* value. This instructs the browser to send the cookie back only on requests to matching directories; a cookie scoped to *domain* of *example.com* and *path* of */some/path/* will be included on a request to

```
http://foo.example.com/some/path/subdirectory/hello_world.txt
```

This mechanism can be deceptive. URL paths are not taken into account during same-origin policy checks and, therefore, do not form a useful security boundary. Regardless of how cookies work, JavaScript code can simply hop between any URLs on a single host at will and inject malicious payloads into

such targets, abusing any functionality protected with path-bound cookies. (Several security books and white papers recommend path scoping as a security measure to this day. In most cases, this advice is dead wrong.)

Other than the true scoping features (which, along with cookie name, constitute a tuple that uniquely identifies every cookie), web servers can also output cookies with two special, independently operated flags: *httponly* and *secure*. The first, *httponly*, prevents access to the cookie via the *document.cookie* API in the hope of making it more difficult to simply copy a user's credentials after successfully injecting a malicious script on a page. The second, *secure*, stops the cookie from being submitted on requests over unencrypted protocols, which makes it possible to build HTTPS services that are resistant to active attacks.[*]

The pitfall of these mechanisms is that they protect data only against reading and not against overwriting. For example, it is still possible for JavaScript code delivered over HTTP to simply overflow the per-domain cookie jar and then set a new cookie without the *secure* flag.[†] Because the *Cookie* header sent by the browser provides no metadata about the origin of a particular cookie or its scope, such a trick is very difficult to detect. A prominent consequence of this behavior is that the common "stateless" way of preventing cross-site request forgery vulnerabilities by simultaneously storing a secret token in a client-side cookie and in a hidden form field, and then comparing the two, is not particularly safe for HTTPS websites. See if you can figure out why!

NOTE *Speaking of destructive interference, until 2010,* httponly *cookies also clashed with* XMLHttpRequest. *The authors of that API simply have not given any special thought to whether the* XMLHttpRequest.getResponseHeader(...) *function should be able to inspect server-supplied* Set-Cookie *values flagged as* httponly*—with predictable results.*

Impact of Cookies on the Same-Origin Policy

The same-origin policy has some undesirable impact on the security of cookies (specifically, on the path-scoping mechanism), but the opposite interaction is more common and more problematic. The difficulty is that HTTP cookies often function as credentials, and in such cases, the ability to obtain them is roughly equivalent to finding a way to bypass SOP. Quite simply, with the right set of cookies, an attacker could use her own browser to interact with the target site on behalf of the victim; same-origin policy is taken out of the picture, and all bets are off.

[*] It does not matter that *https://webmail.example.com/* is offered only over HTTPS. If it uses a cookie that is not locked to encrypted protocols, the attacker may simply wait until the victim navigates to *http://www.fuzzybunnies.com/*, silently inject a frame pointing to *http://webmail.example.com/* on that page, and then intercept the resulting TCP handshake. The browser will then send all the *webmail.example.com* cookies over an unencrypted channel, and at this point the game is essentially over.

[†] Even if this possibility is prevented by separating the jars for *httponly* and normal cookies, multiple identically named but differently scoped cookies must be allowed to coexist, and they will be sent together on any matching requests. They will be not accompanied by any useful metadata, and their ordering will be undefined and browser specific.

Because of this property, any discrepancies between the two security mechanisms can lead to trouble for the more restrictive one. For example, the relatively promiscuous domain-scoping rules used by HTTP cookies mean that it is not possible to isolate fully the sensitive content hosted on *webmail.example.com* from the less trusted HTML present on *blog.example.com*. Even if the owners of the webmail application scope their cookies tightly (usually at the expense of complicating the sign-on process), any attacker who finds a script injection vulnerability on the blogging site can simply overflow the per-domain cookie jar, drop the current credentials, and set his own **.example.com* cookies. These injected cookies will be sent to *webmail.example.com* on all subsequent requests and will be largely indistinguishable from the real ones.

This trick may seem harmless until you realize that such an action may effectively log the victim into a bogus account and that, as a result, certain actions (such as sending email) may be unintentionally recorded within that account and leaked to the attacker before any foul play is noticed. If webmail sounds too exotic, consider doing the same on Amazon or Netflix: Your casual product searches may be revealed to the attacker before you notice anything unusual about the site. (On top of this, many websites are simply not prepared to handle malicious payloads in injected cookies, and unexpected inputs may lead to XSS or similar bugs.)

The antics of HTTP cookies also make it very difficult to secure encrypted traffic against network-level attackers. A *secure* cookie set by *https://webmail.example.com/* can still be clobbered and replaced by a made-up value set by a spoofed page at *http://webmail.example.com/*, even if there is no actual web service listening on port 80 on the target host.

Problems with Domain Restrictions

The misguided notion of allowing domain-level cookies also poses problems for browser vendors and is a continuing source of misery. The key question is how to reliably prevent *example.com* from setting a cookie for **.com* and avoid having this cookie unexpectedly sent to every other destination on the Internet.

Several simple solutions come to mind, but they fall apart when you have to account for country-level TLDs: *example.com.pl* must be prevented from setting a **.com.pl* cookie, too. Realizing this, the original Netscape cookie specification provided the following advice:

> Only hosts within the specified domain can set a cookie for a domain and domains must have at least two (2) or three (3) periods in them to prevent domains of the form: ".com", ".edu", and "va.us".
>
> Any domain that fails within one of the seven special top level domains listed below only requires two periods. Any other domain requires at least three. The seven special top level domains are: "COM", "EDU", "NET", "ORG", "GOV", "MIL", and "INT".

Alas, the three-period rule makes sense only for country-level registrars that mirror the top-level hierarchy (*example.co.uk*) but not for the just as populous group of countries that accept direct registrations (*example.fr*). In fact, there are places where both approaches are allowed; for example, both *example.jp* and *example.co.jp* are perfectly fine.

Because of the out-of-touch nature of this advice, most browsers disregarded it and instead implemented a patchwork of conditional expressions that only led to more trouble. (In one case, for over a decade, you could actually set cookies for **.com.pl*.) Comprehensive fixes to country-code top-level domain handling have shipped in all modern browsers in the past four years, but as of this writing they have not been backported to Internet Explorer 6 and 7, and they probably never will be.

NOTE *To add insult to injury, the Internet Assigned Numbers Authority added a fair number of top-level domains in recent years (for example,* .int *and* .biz*), and it is contemplating a proposal to allow arbitrary generic top-level domain registrations. If it comes to this, cookies will probably have to be redesigned from scratch.*

The Unusual Danger of "localhost"

One immediately evident consequence of the existence of domain-level scoping of cookies is that it is fairly unsafe to delegate any hostnames within a sensitive domain to any untrusted (or simply vulnerable) party; doing so may affect the confidentiality, and invariably the integrity, of any cookie-stored credentials—and, consequently, of any other information handled by the targeted application.

So much is obvious, but in 2008, Tavis Ormandy spotted something far less intuitive and far more hilarious:[5] that because of the port-agnostic behavior of HTTP cookies, an additional danger lies in the fairly popular and convenient administrative practice of adding a "localhost" entry to a domain and having it point to 127.0.0.1.[*] When Ormandy first published his advisory, he asserted that this practice is widespread—not a controversial claim to make—and included the following resolver tool output to illustrate his point:

```
localhost.microsoft.com has address 127.0.0.1
localhost.ebay.com has address 127.0.0.1
localhost.yahoo.com has address 127.0.0.1
localhost.fbi.gov has address 127.0.0.1
localhost.citibank.com has address 127.0.0.1
localhost.cisco.com has address 127.0.0.1
```

Why would this be a security risk? Quite simply, it puts the HTTP services on the user's own machine within the same domain as the remainder of the site, and more importantly, it puts all the services that only *look* like HTTP in the very same bucket. These services are typically not exposed to the Internet, so there is no perceived need to design them carefully or keep them up-to-date. Tavis's case in point is a printer-management service provided by CUPS (Common UNIX Printing System), which would execute attacker-supplied JavaScript in the context of *example.com* if invoked in the following way:

```
http://localhost.example.com:631/jobs/?[...]
  &job_printer_uri=javascript:alert("Hi mom!")
```

[*] This IP address is reserved for loopback interfaces; any attempt to connect to it will route you back to the services running on your own machine.

The vulnerability in CUPS can be fixed, but there are likely many other dodgy local services on all operating systems—everything from disk management tools to antivirus status dashboards. Introducing entries pointing back to 127.0.0.1, or any other destinations you have no control over, ties the security of cookies within your domain to the security of random third-party software. That is a good thing to avoid.

Cookies and "Legitimate" DNS Hijacking

The perils of the domain-scoping policy for cookies don't end with *localhost*. Another unintended interaction is related to the common, widely criticized practice of some ISPs and other DNS service providers of hijacking domain lookups for nonexistent (typically mistyped) hosts. In this scheme, instead of returning the standard-mandated NXDOMAIN response from an upstream name server (which would subsequently trigger an error message in the browser or other networked application), the provider will falsify a record to imply that this name resolves to its site. Its site, in turn, will examine the *Host* header supplied by the browser and provide the user with unsolicited, paid contextual advertising that appears to be vaguely related to her browsing interests. The usual justification offered for this practice is that of offering a more user-friendly browsing experience; the real incentive, of course, is to make more money.

Internet service providers that have relied on this practice include Cablevision, Charter, Earthlink, Time Warner, Verizon, and many more. Unfortunately, their approach is not only morally questionable, but it also creates a substantial security risk. If the advertising site contains any script-injection vulnerabilities, the attacker can exploit them in the context of any other domain simply by accessing the vulnerable functionality through an address such as *nonexistent.example.com*. When coupled with the design of HTTP cookies, this practice undermines the security of any arbitrarily targeted services on the Internet.

Predictably, script-injection vulnerabilities can be found in such hastily designed advertising traps without much effort. For example, in 2008, Dan Kaminsky spotted and publicized a cross-site scripting vulnerability on the pages operated by Earthlink.[6]

All right, all right: It's time to stop obsessing over cookies and move on.

Plug-in Security Rules

Browsers do not provide plug-in developers with a uniform and extensible API for enforcing security policies; instead, each plug-in decides what rules should be applied to executed content and how to put them into action. Consequently, even though plug-in security models are to some extent inspired by the same-origin policy, they diverge from it in a number of ways.

This disconnect can be dangerous. In Chapter 6, we discussed the tendency for plug-ins to rely on inspecting the JavaScript *location* object to determine the origin of their hosting page. This misguided practice forced browser developers to restrict the ability of JavaScript programs to tamper with some

portions of their runtime environment to save the day. Another related, common source of incompatibilities is the interpretation of URLs. For example, in the middle of 2010, one researcher discovered that Adobe Flash had trouble with the following URL:[7]

```
http://example.com:80@bunnyoutlet.com/
```

The plug-in decided that the origin of any code retrieved through this URL should be set to *example.com*, but the browser, when presented with such a URL, would naturally retrieve the data from *bunnyoutlet.com* instead and then hand it over to the confused plug-in for execution.

While this particular bug is now fixed, other vulnerabilities of this type can probably be expected in the future. Replicating some of the URL-parsing quirks discussed in Chapters 2 and 3 can be a fool's errand and, ideally, should not be attempted at all.

It would not be polite to end this chapter on such a gloomy note! Systemic problems aside, let's see how some of the most popular plug-ins approach the job of security policy enforcement.

Adobe Flash

The Flash security model underwent a major overhaul in 2008,[8] and since then, it has been reasonably robust. Every loaded Flash applet is now assigned an SOP-like origin derived from its originating URL* and is granted nominal origin-related permissions roughly comparable to those of JavaScript. In particular, each applet can load cookie-authenticated content from its originating site, load some constrained datatypes from other origins, and make same-origin *XMLHttpRequest*-like HTTP calls through the *URLRequest* API. The set of permissible methods and request headers for this last API is managed fairly reasonably and, as of this writing, is more restrictive than most of the browser-level blacklists for *XMLHttpRequest* itself.[9]

On top of this sensible baseline, three flexible but easily misused mechanisms permit this behavior to be modified to some extent, as discussed next.

Markup-Level Security Controls

The embedding page can specify three special parameters provided through *<embed>* or *<object>* tags to control how an applet will interact with its host page and the browser itself:

- ***AllowScriptAccess* parameter** This setting controls an applet's ability to use the JavaScript *ExternalInterface.call(...)* bridge (see Chapter 8) to execute JavaScript statements in the context of the embedding site. Possible values are *always*, *never*, and *sameorigin*; the last setting gives access to the page only if the page is same-origin with the applet itself. (Prior to the 2008 security overhaul, the plug-in defaulted to *always*; the current default is the much safer *sameorigin*.)

* In some contexts, Flash may implicitly permit access from HTTPS origins to HTTP ones but not the other way round. This is usually harmless, and as such, it is not given special attention throughout the remainder of this section.

- ***AllowNetworking* parameter** This poorly named setting restricts an applet's permission to open or navigate browser windows and to make HTTP requests to its originating server. When set to *all* (the default), the applet can interfere with the browser; when set to *internal*, it can perform only nondisruptive, internal communications through the Flash plug-in. Setting this parameter to *none* disables most network-related APIs altogether.* (Prior to recent security improvements, *allowNetworking=all* opened up several ways to bypass *allowScriptAccess=none*, for example, by calling *getURL(...)* on a *javascript:* URL. As of this writing, however, all scripting URLs should be blacklisted in this scenario.)
- ***AllowFullScreen* parameter** This parameter controls whether an applet should be permitted to go into full-screen rendering mode. The possible values are *true* and *false*, with *false* being the default. As noted in Chapter 8, the decision to give this capability to Flash applets is problematic due to UI spoofing risks; it should be not enabled unless genuinely necessary.

Security.allowDomain(...)

The *Security.allowDomain(...)* method[10] allows Flash applets to grant access to their variables and functions to any JavaScript code or to other applets coming from a different origin. Buyer beware: Once such access is granted, there is no reliable way to maintain the integrity of the original Flash execution context. The decision to grant such permissions should not be taken lightly, and the practice of calling *allowDomain("*")* should usually be punished severely.

Note that a weirdly named *allowInsecureDomain(...)* method is also available. The existence of this method does not indicate that *allowDomain(...)* is particularly secure; rather, the "insecure" variant is provided for compatibility with ancient, pre-2003 semantics that completely ignored the HTTP/HTTPS divide.

Cross-Domain Policy Files

Through the use of *loadPolicyFile(...)*, any Flash applet can instruct its runtime environment to retrieve a security policy file from an almost arbitrary URL. This XML-based document, usually named *crossdomain.xml*, will be interpreted as an expression of consent to cross-domain, server-level access to the origin determined by examining the policy URL.[11] The syntax of a policy file is fairly self-explanatory and may look like this:

```
<cross-domain-policy>
  <allow-access-from domain="foo.example.com"/>
  <allow-http-request-headers-from domain="*.example.com"
     headers="X-Some-Header" />
</cross-domain-policy>
```

* It should not be assumed that this setting prevents any sensitive data available to a rogue applet from being relayed to third parties. There are many side channels that any Flash applet could leverage to leak information to a cooperating party without directly issuing network requests. In the simplest and most universal case, CPU loads can be manipulated to send out individual bits of information to any simultaneously loaded applet that continuously samples the responsiveness of its runtime environment.

The policy may permit actions such as loading cross-origin resources or issuing arbitrary *URLRequest* calls with whitelisted headers, through the browser HTTP stack. Flash developers do attempt to enforce a degree of path separation: A policy loaded from a particular subdirectory can in principle permit access only to files within that path. In practice, however, the interactions with SOP and with various path-mapping semantics of modern browsers and web application frameworks make it unwise to depend on this boundary.

NOTE *Making raw TCP connections via* XMLSocket *is also possible and controlled by an XML policy, but following Flash's 2008 overhaul,* XMLSocket *requires that a separate policy file be delivered on TCP port 843 of the destination server. This is fairly safe, because no other common services run on this port and, on many operating systems, only privileged users can launch services on any port below 1024. Because of the interactions with certain firewall-level mechanisms, such as FTP protocol helpers, this design may still cause some network-level interference,*[12] *but this topic is firmly beyond the scope of this book*

As expected, poorly configured *crossdomain.xml* policies are an appreciable security risk. In particular, it is a very bad idea to specify *allow-access-from* rules that point to any domain you do not have full confidence in. Further, specifying "*" as a value for this parameter is roughly equivalent to executing *document.domain = "com"*. That is, it's a death wish.

Policy File Spoofing Risks

Other than the possibility of configuration mistakes, another security risk with Adobe's policy-based security model is that random user-controlled documents may be interpreted as cross-domain policies, contrary to the site owner's intent.

Prior to 2008, Flash used a notoriously lax policy parser, which when processing *loadPolicyFile(...)* files would skip arbitrary leading garbage in search of the opening <*cross-domain-policy*> tag. It would simply ignore the MIME type returned by the server when downloading the resource, too. As a result, merely hosting a valid, user-supplied JPEG image could become a grave security risk. The plug-in also skipped over any HTTP redirects, making it dangerous to do something as simple as issuing an HTTP redirect to a location you did not control (an otherwise harmless act).

Following the much-needed revamp of the *loadPolicyFile* behavior, many of the gross mistakes have been corrected, but the defaults are still not perfect. On the one hand, redirects now work intuitively, and the file must be a well-formed XML document. On the other, permissible MIME types include *text/**, *application/xml*, and *application/xhtml+xml*, which feels a bit too broad. *text/plain* or *text/csv* may be misinterpreted as a policy file, and that should not be the case.

Thankfully, to mitigate the problem, Adobe engineers decided to roll out *meta-policies*, policies that are hosted at a predefined, top-level location (*/crossdomain.xml*) that the attacker can't override. A meta-policy can specify sitewide restrictions for all the remaining policies loaded from attacker-supplied

URLs. The most important of these restrictions is *<site-control permitted-cross-domain-policies="...">*. This parameter, when set to *master-only*, simply instructs the plug-in to disregard subpolicies altogether. Another, less radical value, *by-content-type*, permits additional policies to be loaded but requires them to have a nonambiguous *Content-Type* header set to *text/x-cross-domain-policy*.

Needless to say, it's highly advisable to use a meta-policy that specifies one of these two directives.

Microsoft Silverlight

If the transition from Flash to Silverlight seems abrupt, it's because the two are easy to confuse. The Silverlight plug-in borrows from Flash with remarkable zeal; in fact, it is safe to say that most of the differences between their security models are due solely to nomenclature. Microsoft's platform uses the same-origin-determination approach, substitutes *allowScriptAccess* with *enableHtmlAccess*, replaces *crossdomain.xml* with the slightly different *clientaccesspolicy.xml* syntax, provides a *System.Net.Sockets* API instead of *XMLSocket*, uses *HttpWebRequest* in place of *URLRequest*, rearranges the flowers, and changes the curtains in the living room.

The similarities are striking, down to the list of blocked request headers for the *HttpWebRequest* API, which even includes *X-Flash-Version* from the Adobe spec.[13] Such consistency is not a problem, though: In fact, it is preferable to having a brand-new security model to take into account. Plus, to its credit, Microsoft did make a couple of welcome improvements, including ditching the insecure *allowDomain* logic in favor of *RegisterScriptableObject*, an approach that allows only explicitly specified callbacks to be exposed to third-party domains.

Java

Sun's Java (now officially belonging to Oracle) is a very curious case. Java is a plug-in that has fallen into disuse, and its security architecture has not received much scrutiny in the past decade or so. Yet, because of its large installed base, it is difficult to simply ignore it and move on.

Unfortunately, the closer you look, the more evident it is that the ideas embraced by Java tend to be incompatible with the modern Web. For example, a class called *java.net.HttpURLConnection*[14] permits credential-bearing HTTP requests to be made to an applet's originating website, but the "originating website" is understood as *any* website hosted at a particular IP address, as sanctioned by the *java.net.URL.equals(...)* check. This model essentially undoes any isolation between HTTP/1.1 virtual hosts—an isolation strongly enforced by the same-origin policy, HTTP cookies, and virtually all other browser security mechanisms in use today.

Further along these lines, the *java.net.URLConnection* class[15] allows arbitrary request headers, including *Host*, to be set by the applet, and another class, *Socket*,[16] permits unconstrained TCP connections to arbitrary ports on the originating server. All of these behaviors are frowned upon in the browser and in any other contemporary plug-in.

Origin-agnostic access from the applet to the embedding page is provided through the *JSObject* mechanism and is expected to be controlled by the embedding party through the *mayscript* attribute specified in the <*applet*>, <*embed*>, or <*object*> tags.[17] The documentation suggests that this is a security feature:

> Due to security reasons, JSObject support is not enabled in Java Plug-in by default. To enable JSObject support in Java Plug-in, a new attribute called MAYSCRIPT needs to be present in the EMBED/OBJECT tag.

Unfortunately, the documentation neglects to mention that another closely related mechanism, *DOMService*,[18] ignores this setting and gives applets largely unconstrained access to the embedding page. While *DOMService* is not supported in Firefox and Opera, it is available in other browsers, which makes any attempt to load third-party Java content equivalent to granting full access to the embedding site.

Whoops.

NOTE *Interesting fact: Recent versions of Java attempt to copy the* crossdomain.xml *support available in Flash.*

Coping with Ambiguous or Unexpected Origins

This concludes our overview of the basic security policies and consent isolation mechanisms. If there is one observation to be made, it's that most of these mechanisms depend on the availability of a well-formed, canonical hostname from which to derive the context for all the subsequent operations. But what if this information is not available or is not presented in the expected form?

Well, that's when things get funny. Let's have a look at some of the common corner cases, even if just for fleeting amusement.

IP Addresses

Due to the failure to account for IP addresses when designing HTTP cookies and the same-origin policy, almost all browsers have historically permitted documents loaded from, say, *http://1.2.3.4/* to set cookies for a "domain" named **.3.4*. Adjusting *document.domain* in a similar manner would work as well. In fact, some of these behaviors are still present in older versions of Internet Explorer.

This behavior is unlikely to have an impact on mainstream web applications, because such applications are not meant to be accessed through an IP-based URL and will often simply fail to function properly. But a handful of systems, used primarily by technical staff, are meant to be accessed by their IP addresses; these systems may simply not have DNS records configured at all. In these cases, the ability for *http://1.2.3.4/* to inject cookies for *http://123.234.3.4/* may be an issue. The IP-reachable administrative interfaces of home routers are of some interest, too.

Hostnames with Extra Periods

At their core, cookie-setting algorithms still depend on counting the number of periods in a URL to determine whether a particular *domain* parameter is acceptable. In order to make the call, the count is typically correlated with a list of several hundred entries on the vendor-maintained Public Suffix List (*http://publicsuffix.org/*).

Unfortunately for this algorithm, it is often possible to put extra periods in a hostname and still have it resolve correctly. Noncanonical hostname representations with excess periods are usually honored by OS-level resolvers and, if honored, will confuse the browser. Although said browser would not automatically consider a domain such as *www.example.com.pl.* (with an extra trailing period) to be the same as the real *www.example.com.pl*, the subtle and seemingly harmless difference in the URL could escape even the most attentive users.

In such a case, interacting with the URL with trailing period can be unsafe, as other documents sharing the **.com.pl.* domain may be able to inject cross-domain cookies with relative ease.

This period-counting problem was first noticed around 1998.[19] About a decade later, many browser vendors decided to roll out basic mitigations by adding a yet another special case to the relevant code; as of this writing, Opera is still susceptible to this trick.

Non–Fully Qualified Hostnames

Many users browse the Web with their DNS resolvers configured to append local suffixes to all found hostnames, often without knowing. Such settings are usually sanctioned by ISPs or employers through automatic network configuration data (Dynamic Host Configuration Protocol, DHCP).

For any user browsing with such a setting, the resolution of DNS labels is ambiguous. For example, if the DNS search path includes *coredump.cx*, then *www.example.com* may resolve to the real *www.example.com* website or to *www.example.com.coredump.cx* if such a record exists. The outcomes are partly controlled by configuration settings and, to some extent, can be influenced by an attacker.

To the browser, both locations appear to be the same, which may have some interesting side effects. Consider one particularly perverse case: Should *http://com*, which actually resolves to *http://com.coredump.cx/*, be able to set **.com* cookies by simply omitting the *domain* parameter?

Local Files

Because local resources loaded through the *file:* protocol do not have an explicit hostname associated with them, it's impossible for the browser to compute a normal origin. For a very long time, the vendors simply decided that the best course of action in such a case would be to simply ditch the same-origin policy. Thus, any HTML document saved to disk would automatically

be granted access to any other local files via *XMLHttpRequest* or DOM and, even more inexplicably, would be able to access any Internet-originating content in the same way.

This proved to be a horrible design decision. No one expected that the mere act of downloading an HTML document would put all of the user's local files, and his online credentials, in jeopardy. After all, accessing that same document over the Web would be perfectly safe.

Many browsers have tried to close this loophole in recent years, with varying degrees of success:

Chrome (and, by extension, other WebKit browsers)
The Chrome browser completely disallows any cross-document DOM or *XMLHttpRequest* access from *file:* origins, and it ignores *document.cookie* calls or *<meta http-equiv="Set-Cookie" ...>* directives in this setting. Access to a *localStorage* container shared by all *file:* documents is permitted, but this may change soon.

Firefox
Mozilla's browser permits access only to files within the directory of the original document, as well as nearby subdirectories. This policy is pretty good, but it still poses some risk to documents stored or previously downloaded to that location. Access to cookies via *document.cookie* or *<meta http-equiv="Set-Cookie" ...>* is possible, and all *file:* cookies are visible to any other local JavaScript code.* The same holds true for access to storage mechanisms.

Internet Explorer 7 and above
Unconstrained access to local and Internet content from *file:* origins is permitted, but it requires the user to click through a nonspecific warning to execute JavaScript first. The consequences of this action are not explained clearly (the help subsystem cryptically states that *"Internet Explorer restricts this content because occasionally these programs can malfunction or give you content you don't want"*), and many users may well be tricked into clicking through the prompt.

Internet Explorer's cookie semantics are similar to those of Firefox. Web storage is not supported in this origin, however.

Opera and Internet Explorer 6
Both of these browsers permit unconstrained DOM or *XMLHttpRequest* access without further checks. Noncompartmentalized *file:* cookies are permitted, too.

NOTE *Plug-ins live by their own rules in* file: *land: Flash uses a* local-with-filesystem *sandbox model,[20] which gives largely unconstrained access to the local filesystem, regardless of the policy enforced by the browser itself, while executing Java or Windows Presentation Framework applets from the local filesystem may in some cases be roughly equivalent to running an untrusted binary.*

* Because there is no compartmentalization between *file:* cookies, it is unsafe to rely on them for legitimate purposes. Some locally installed HTML applications ignore this advice, and consequently, their cookies can be easily tampered with by any downloaded, possibly malicious, HTML document viewed by the user.

Pseudo-URLs

The behavior of pseudo-URLs such as *about:*, *data:*, or *javascript:* originally constituted a significant loophole in the implementations of the same-origin policy. All such URLs would be considered same-origin and would permit unconstrained cross-domain access from any other resource loaded over the same scheme. The current behavior, which is very different, will be the topic of the next chapter of this book; in a nutshell, the status quo reflects several rounds of hastily implemented improvements and is a complex mix of browser-specific special cases and origin-inheritance rules.

Browser Extensions and UI

Several browsers permit JavaScript-based UI elements or certain user-installed browser extensions to run with elevated privileges. These privileges may entail circumventing specific SOP checks or calling normally unavailable APIs in order to write files, modify configuration settings, and so on.

Privileged JavaScript is a prominent feature of Firefox, where it is used with XUL to build large portions of the browser user interface. Chrome also relies on privileged JavaScript to a smaller but still notable degree.

The same-origin policy does not support privileged contexts in any specific way. The actual mechanism by which extra privileges are granted may involve loading the document over a special and normally unreachable URL scheme, such as *chrome:* or *res:*, and then adding special cases for that scheme in other portions of the browser code. Another option is simply to toggle a binary flag for a JavaScript context, regardless of its actual origin, and examine that flag later. In all cases, the behavior of standard APIs such as *localStorage*, *document.domain*, or *document.cookie* may be difficult to predict and should not be relied upon: Some browsers attempt to maintain isolation between the contexts belonging to different extensions, but most don't.

NOTE *Whenever writing browser extensions, any interaction with nonprivileged contexts must be performed with extreme caution. Examining untrusted contexts can be difficult, and the use of mechanisms such as* eval(...) *or* innerHMTL *may open up privilege-escalation paths.*

Other Uses of Origins

Well, that's all to be said about browser-level content isolation logic for now. It is perhaps worth noting that the concept of origins and host- or domain-based security mechanisms is not limited to that particular task and makes many other appearances in the browser world. Other quasi-origin-based privacy or security features include preferences and cached information related to per-site cookie handling, pop-up blocking, geolocation sharing, password management, camera and microphone access (in Flash), and much, much more. These features tend to interact with the security features described in this chapter at least to some extent; we explore this topic in more detail soon.

Security Engineering Cheat Sheet

Good Security Policy Hygiene for All Websites

To protect your users, include a top-level *crossdomain.xml* file with the *permitted-cross-domain-policies* parameter set to *master-only* or *by-content-type*, even if you do not use Flash anywhere on your site. Doing so will prevent unrelated attacker-controlled content from being misinterpreted as a secondary *crossdomain.xml* file, effectively undermining the assurances of the same-origin policy in Flash-enabled browsers.

When Relying on HTTP Cookies for Authentication

- ☑ Use the *httponly* flag; design the application so that there is no need for JavaScript to access authentication cookies directly. Sensitive cookies should be scoped as tightly as possible, preferably by not specifying *domain* at all.
- ☑ If the application is meant to be HTTPS only, cookies must be marked as *secure*, and you must be prepared to handle cookie injection gracefully. (HTTP contexts may overwrite *secure* cookies, even though they can't read them.) Cryptographic cookie signing may help protect against unconstrained modification, but it does not defend against replacing a victim's cookies with another set of legitimately obtained credentials.

When Arranging Cross-Domain Communications in JavaScript

- ☑ Do not use *document.domain*. Rely on *postMessage(...)* where possible and be sure to specify the destination origin correctly; then verify the sender's origin when receiving the data on the other end. Beware of naïve substring matches for domain names: *msg.origin.indexOf(".example.com")* is very insecure.
- ☑ Note that various pre-*postMessage* SOP bypass tricks, such as relying on *window.name*, are not tamper-proof and should not be used for exchanging sensitive data.

When Embedding Plug-in-Handled Active Content from Third Parties

Consult the cheat sheet in Chapter 8 first for general advice.

- ☑ **Flash:** Do not specify *allowScriptAccess=always* unless you fully trust the owner of the originating domain and the security of its site. Do not use this setting when embedding HTTP applets on HTTPS pages. Also, consider restricting *allowFullScreen* and *allowNetworking* as appropriate.
- ☑ **Silverlight:** Do not specify *enableHtmlAccess=true* unless you trust the originating domain, as above.
- ☑ **Java:** Java applets can't be safely embedded from untrusted sources. Omitting *mayscript* does not fully prevent access to the embedding page, so do not attempt to do so.

When Hosting Your Own Plug-in-Executed Content

☑ Note that many cross-domain communication mechanisms provided by browser plug-ins may have unintended consequences. In particular, avoid *crossdomain.xml*, *clientaccesspolicy.xml*, or *allowDomain(...)* rules that point to domains you do not fully trust.

When Writing Browser Extensions

☑ Avoid relying on *innerHTML*, *document.write(...)*, *eval(...)*, and other error-prone coding patterns, which can cause code injection on third-party pages or in a privileged JavaScript context.

☑ Do not make security-critical decisions by inspecting untrusted JavaScript security contexts, as their behavior can be deceptive.

10

ORIGIN INHERITANCE

Some web applications rely on pseudo-URLs such as *about:*, *javascript:*, or *data:* to create HTML documents that do not contain any server-supplied content and that are instead populated with the data constructed entirely on the client side. This approach eliminates the delay associated with the usual HTTP requests to the server and results in far more responsive user interfaces.

Unfortunately, the original vision of the same-origin policy did not account for such a use case. Specifically, a literal application of the protocol-, host-, and port-matching rules discussed in Chapter 9 would cause every *about:blank* document created on the client side to have a different origin from its parent page, preventing it from being meaningfully manipulated. Further, all *about:blank* windows created by completely unrelated websites would belong to the same origin and, under the right circumstances, would be able to interfere with each other with no supervision at all.

To address this incompatibility of client-side documents with the same-origin policy, browsers gradually developed incompatible and sometimes counterintuitive approaches to computing a synthetic origin and access permissions for pseudo-URLs. An understanding of these rules is important on its own merit, and it will lay the groundwork for the discussion of certain other SOP exceptions in Chapter 11.

Origin Inheritance for about:blank

The *about:* scheme is used in modern browsers for a variety of purposes, most of which are not directly visible to normal web pages. The *about:blank* document is an interesting special case, however: This URL can be used to create a minimal DOM hierarchy (essentially a valid but empty document) to which the parent document may write arbitrary data later on.

Here is an example of a typical use of this scheme:

```
<iframe src="about:blank" name="test"></iframe>

<script>
  ...
  frames["test"].document.body.innerHTML = "<h1>Hi mom!</h1>";
  ...
</script>
```

NOTE *In the HTML markup provided in this example, and when creating new windows or frames in general,* about:blank *can be omitted. The value is defaulted to when no other URL is specified by the creator of the parent document.*

In every browser, most types of navigation to *about:blank* result in the creation of a new document that inherits its SOP origin from the page that initiated the navigation. The inherited origin is reflected in the *document.domain* property of the new JavaScript execution context, and DOM access to or from any other origins is not permitted.

This simple formula holds true for navigation actions such as clicking a link, submitting a form, creating a new frame or a window from a script, or programmatically navigating an existing document. That said, there are exceptions, the most notable of which are several special, user-controlled navigation methods. These include manually entering *about:blank* in the address bar, following a bookmark, or performing a gesture reserved for opening a link in a new window or a tab.[*] These actions will result in a document that occupies a unique synthetic origin and that can't be accessed by any other page.

Another special case is the loading of a normal server-supplied document that subsequently redirects to *about:blank* using *Location* or *Refresh*. In Firefox and WebKit-based browsers, such redirection results in a unique, non-accessible origin, similar to the scenario outlined in the previous paragraph. In Internet Explorer, on the other hand, the resulting document will be

[*] This is usually accomplished by holding CTRL or SHIFT while clicking on a link, or by right-clicking the mouse to access a contextual menu, and then selecting the appropriate option.

accessible by the parent page if the redirection occurs inside an *<iframe>* but not if it took place in a separate window. Opera's behavior is the most difficult to understand: *Refresh* results in a document that can be accessed by the parent page, but the *Location* redirect will give the resulting page the origin of the site that performed the redirect.

Further, it is possible for a parent document to navigate an existing document frame to an *about:blank* URL, even if the existing document shown in that container has a different origin than the caller.* The newly created blank document will inherit the origin from the caller in all browsers other than Internet Explorer. In the case of Internet Explorer, such navigation will succeed but will result in an inaccessible document. (This behavior is most likely not intentional.)

If this description makes your head spin, the handling of *about:blank* documents is summarized in Table 10-1.

Table 10-1: Origin Inheritance for *about:blank* URLs

	Type of navigation				
	New page	**Existing non-same-origin page**	***Location* redirect**	***Refresh* redirect**	**URL entry or gesture**
Internet Explorer	Inherited from caller	Unique origin	(Denied)	**Frame**: Inherited from parent	Unique origin
				Window: Unique origin	
Firefox	Inherited from caller		Unique origin		
All WebKit	Inherited from caller		(Denied)	Unique origin	
Opera	Inherited from caller		Inherited from redirecting party	Inherited from parent	

Inheritance for data: URLs

The *data:* scheme,[1] first outlined in Chapter 2, was designed to permit small documents, such as icons, to be conveniently encoded and then directly inlined in an HTML document, saving time on HTTP round-trips. For example:

```
<img src="...">
```

When the *data:* scheme is used in conjunction with type-specific subresources, the only unusual security consideration is that it poses a challenge for plug-ins that wish to derive permissions for an applet from its originating

* The exact circumstances that make this possible will be the focus of Chapter 11. For now, suffice it to say that this can be accomplished in many settings in a browser-specific way. For example, in Firefox, you call *window.open(..., 'target')*, while in Internet Explorer, calling *target.location.assign(...)* is the way to go.

URL. The origin can't be computed by looking at the URL alone, and the behavior is somewhat unpredictable and highly plug-in specific (for example, Adobe Flash currently rejects any attempts to use *data:* documents).

More important than the case of type-specific content is the use of *data:* as a destination for windows and frames. In all browsers but Internet Explorer, the scheme can be used as an improved variant of *about:blank*, as in this example:

```
<iframe src="data:text/html;charset=utf-8,<h1>Hi mom!</h1>">
</iframe>
```

In this scenario, there is no compelling reason for a *data:* URL to behave differently than *about:blank*. In reality, however, it will behave differently in some browsers and therefore must be used with care.

- **WebKit browsers** In Chrome and Safari, all *data:* documents are given a unique, nonaccessible origin and do not inherit from the parent at all.
- **Firefox** In Firefox, the origin for *data:* documents is inherited from the navigating context, similar to *about:blank*. However, unlike with *about:blank*, manually entering *data:* URLs or opening bookmarked ones results in the new document inheriting origin from the page on which the navigation occurred.
- **Opera** As of this writing, a shared "empty" origin is used for all *data:* URLs, which is accessible by the parent document. This approach is unsafe, as it may allow cross-domain access to frames created by unrelated pages, as shown in Figure 10-1. (I reported this behavior to Opera, and it likely will be amended soon.)
- **Internet Explorer** *data:* URLs are not supported in Internet Explorer versions prior to 8. The scheme is supported only for select types of subresources in Internet Explorer 8 and 9 and can't be used for navigation.

Table 10-2 summarizes the current behavior of *data:* URLs.

Table 10-2: Origin Inheritance for *data:* URLs

	Type of navigation				
	New page	**Existing non-same-origin page**	***Location* redirect**	***Refresh* redirect**	**URL entry or gesture**
Internet Explorer 6/7	(Not supported)				
Internet Explorer 8/9	(Not supported for navigation)				
Firefox	Inherited from caller		Unique origin		Inherited from previous page
All WebKit	Unique origin		(Denied)	Unique origin	Unique origin
Opera	Shared origin (This is a bug!)		(Denied)	Inherited from parent	

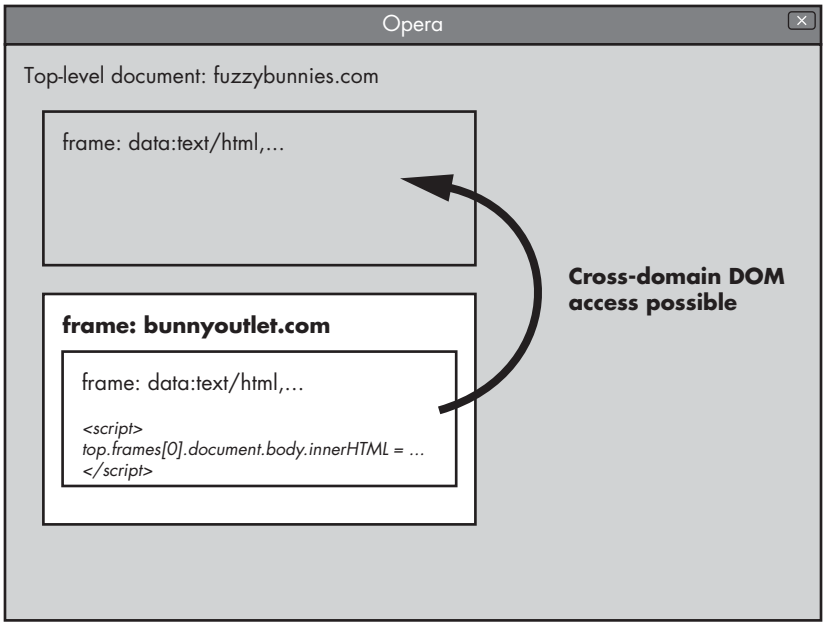


Figure 10-1: Access between data: *URLs in Opera*

Inheritance for javascript: and vbscript: URLs

Scripting-related pseudo-URLs, such as *javscript:*, are a very curious mechanism. Using them to load some types of subresources will lead to code execution in the context of the document that attempts to load such an operation (subject to some inconsistent restrictions, as discussed in Chapter 4). An example of this may be

```
<iframe src="javascript:alert('Hi mom!')"></iframe>
```

More interestingly (and far less obviously) than the creation of new subresources, navigating existing windows or frames to *javascript:* URLs will cause the inlined JavaScript code to execute in the context of the navigated page (and not the navigating document!)—even if the URL is entered manually or loaded from a bookmark.

Given this behavior, it is obviously very unsafe to allow one document to navigate any other non-same-origin context to a *javascript:* URL, as it would enable the circumvention of all other content-isolation mechanisms: Just load *fuzzybunnies.com* in a frame, and then navigate that frame to *javascript:do_evil_stuff()* and call it a day. Consequently, such navigation is prohibited in all browsers except for Firefox. Firefox appears to permit it for some reason, but it changes the semantics in a sneaky way. When the origin of the caller and the navigation target do not match, it executes the *javascript:* payload in a special null origin, which lacks its own DOM or any of the browser-supplied I/O functions registered (thus permitting only purely algorithmic operations to occur).

The cross-origin case is dangerous, but its same-origin equivalent is not: Within a single origin, any content is free to navigate itself or its peers to *javascript:* URLs on its own volition. In this case, the *javascript:* scheme is honored when following links, submitting forms, calling *location.assign(...)*, and so on. In WebKit and Opera, *Refresh* redirection to *javascript:* will work as well; other browsers reject such navigation due to vague and probably misplaced script-injection concerns.

The handling of scripting URLs is outlined in Table 10-3.

Table 10-3: Origin Inheritance for Scripting URLs

	Type of navigation					
	New page	**Existing same-origin page**	**Existing non-same-origin page**	***Location* redirect**	***Refresh* redirect**	**URL entry or gesture**
Internet Explorer	Inherited from caller	Inherited from navigated page	(Denied)	(Denied)	(Denied)	Inherited from navigated page
Firefox			Null context		(Denied)	
All WebKit			(Denied)		Inherited from navigated page	
Opera			(Denied)		Inherited from navigated page	

On top of these fascinating semantics, there is a yet another twist unique to the *javascript:* scheme: In some cases, the handling of such script-containing URLs involves a second step. Specifically, if the supplied code evaluates properly, and the value of the last statement is nonvoid and can be converted to a string, this string will be interpreted as an HTML document and will replace the navigated page (inheriting origin from the caller). The logic governing this curious behavior is very similar to that influencing the behavior of *data:* URLs. An example of such a document-replacing expression is this:

```
javascript:"<b>2 + 2 = " + (2+2) + "</b>"
```

A Note on Restricted Pseudo-URLs

The somewhat quirky behavior of the three aforementioned classes of URLs—*about:blank*, *javascript:*, and *data:*—are all that most websites need to be concerned with. Nevertheless, browsers use a range of other documents with no inherent, clearly defined origin (e.g., *about:config* in Firefox, a privileged JavaScript page that can be used to tweak the browser's various under-the-hood settings, or *chrome://downloads* in Chrome, which lists the recently downloaded documents with links to open any of them). These documents are a continued source of security problems, even if they are not reachable directly from the Internet.

Because of the incompatibility of these URLs with the boundaries controlled by the same-origin policy, special care must be taken to make sure that these URLs are sufficiently isolated from other content whenever they are loaded in the browser as a result of user action or some other indirect browser-level process. An interesting case illustrating the risk is a 2010 bug in the way Firefox handled *about:neterror*.[2] Whenever Firefox can't correctly retrieve a document from a remote server (a condition that is usually easy to trigger with a carefully crafted link), it puts the destination URL in the address bar but loads *about:neterror* in place of the document body. Unfortunately, due to a minor oversight, this special error page would be same-origin with any *about:blank* document opened by any Internet-originating content, thereby permitting the attacker to inject arbitrary content into the *about:neterror* window while preserving the displayed destination URL.

The moral of this story? Avoid the urge to gamble with the same-origin policy; instead, play along with it. Note that making *about:neterror* a hierarchical URL, instead of trying to keep track of synthetic origins, would have prevented the bug.

Security Engineering Cheat Sheet

Because of their incompatibility with the same-origin policy, *data:*, *javascript:*, and implicit or explicit *about:blank* URLs should be used with care. When performance is not critical, it is preferable to seed new frames and windows by pointing them to a server-supplied blank document with a definite origin first.

Keep in mind that *data:* and *javascript:* URLs are not a drop-in replacement for *about:blank*, and they should be used only when absolutely necessary. In particular, it is currently unsafe to assume that *data:* windows can't be accessed across domains.

11

LIFE OUTSIDE SAME-ORIGIN RULES

The same-origin policy is the most important mechanism we have to keep hostile web applications at bay, but it's also an imperfect one. Although it is meant to offer a robust degree of separation between any two different and clearly identifiable content sources, it often fails at this task.

To understand this disconnect, recall that contrary to what common sense may imply, the same-origin policy was never meant to be all-inclusive. Its initial focus, the DOM hierarchy (that is, just the *document* object exposed to JavaScript code) left many of the peripheral JavaScript features completely exposed to cross-domain manipulation, necessitating ad hoc fixes. For example, a few years after the inception of SOP, vendors realized that allowing third-party documents to tweak the *location.host* property of an unrelated window is a bad idea and that such an operation could send potentially sensitive data present in other URL segments to an attacker-specified site. The policy has

subsequently been extended to at least partly protect this and a couple of other sensitive objects, but in some less clear-cut cases, awkward loopholes remain.

The other problem is that many cross-domain interactions happen completely outside of JavaScript and its object hierarchy. Actions such as loading third-party images or stylesheets are deeply rooted in the design of HTML and do not depend on scripting in any meaningful way. (In principle, it would be possible to retrofit them with origin-based security controls, but doing so would interfere with existing websites. Plus, some think that such a decision would go against the design principles that made the Web what it is; they believe that the ability to freely cross-reference content should not be infringed upon.)

In light of this, it seems prudent to explore the boundaries of the same-origin policy and learn about the rich life that web applications can lead outside its confines. We begin with document navigation—a mechanism that at first seems strikingly simple but that is really anything but.

Window and Frame Interactions

On the Web, the ability to steer the browser from one website to another is taken for granted. Some of the common methods of achieving such navigation are discussed throughout Part I of this book; the most notable of these are HTML links, forms, and frames; HTTP redirects; and JavaScript *window.open(…)* and *location.** calls.

Actions such as pointing a newly opened window to an off-domain URL or specifying the *src* parameter of a frame are intuitive and require no further review. But when we look at the ability of one page to navigate another, existing document—well, the reign of intuition comes to a sudden end.

Changing the Location of Existing Documents

In the simple days before the advent of HTML frames, only one document could occupy a given browser window, and only that single window would be under the document's control. Frames changed this paradigm, however, permitting several different and completely separate documents to be spliced into a single logical view, coexisting within a common region of the screen. The introduction of the mechanism also necessitated another step: To sanely implement certain frame-based websites, any of the component documents displayed in a window needed the ability to navigate its neighboring frames or perhaps the top-level document itself. (For example, imagine a two-frame page with a table of contents on the left and the actual chapter on the right. Clicking a chapter name in the left pane should navigate the chapter in the right pane, and nothing else.)

The mechanism devised for this last purpose is fairly simple: One can specify the *target* parameter on *<a href=…>* links or forms, or provide the name of a window to the JavaScript method known as *window.open(…)*, in

order to navigate any other, previously named document view. In the mid-1990s, when this functionality first debuted, there seemed to be no need to incorporate any particular security checks into this logic; any page could navigate any other named window or a frame displayed by the browser to a new location at will.

To understand the consequences of this design, it is important to pause for a moment and examine the circumstances under which a particular document may obtain a name to begin with. For frames, the story is simple: In order to reference a frame easily on the embedding page, virtually all frames have a *name* attribute (and some browsers, such as Chrome, also look at *id*). Browser windows, on the other hand, are typically anonymous (that is, their *window.name* property is an empty string), unless created programmatically; in the latter case, the name is specified by whoever creates the view. Anonymous windows do not necessarily stay anonymous, however. If a rogue application is displayed in such a window even briefly, it may set the *window.name* property to any value, and this effect will persist.

The aforementioned ability to target windows and frames by name is not the only way to navigate them; JavaScript programs that hold window handles pointing to other documents may directly invoke certain DOM methods without knowing the name of their target at all. Attacker-supplied code will not normally hold handles to completely unrelated windows, but it can traverse properties such as *opener*, *top*, *parent*, or *frames[]* in order to locate even distant relatives within the same navigation flow. An example of such a far-reaching lookup (and subsequently, navigation) is

```
opener.opener.frames[2].location.assign("http://www.bunnyoutlet.com/");
```

These two lookup techniques are not mutually exclusive: JavaScript programs can first obtain the handle of an unrelated but named window through *window.open(…)* and then traverse the *opener* or *frames[]* properties of that context in order to reach its interesting relatives nearby.

Once a suitable handle is looked up in any fashion, the originating context can leverage one of several DOM methods and properties in order to change the address of the document displayed in that view. In every contemporary browser, calling the *<handle>.location.replace(…)* method, or assigning a value to *<handle>.location* or *<handle>.location.href* properties, should do the trick. Amusingly, due to random implementation quirks, other theoretically equivalent approaches (such as invoking *<handle>.location.assign(…)* or *<handle>.window.open(…, "_self")*) may be hit-and-miss.

Okay, so it may be possible to navigate unrelated documents to new locations—but let's see what could possibly go wrong.

Frame Hijacking Risks

The ability for one domain to navigate windows created by other sites, or ones that are simply no longer same-origin with their creator, is usually not a grave concern. This laid-back design may be an annoyance and may pose

some minor, speculative phishing risk,* but in the grand scheme of things, it is neither a very pronounced issue nor a particularly distinctive one. This is, perhaps, the reason why the original authors of the relevant APIs have not given the entire mechanism too much thought.

Alas, the concept of HTML frames alters the picture profoundly: Any application that relies on frames to build a trusted user interface is at an obvious risk if an unrelated site is permitted to hijack such UI elements without leaving any trace of the attack in the address bar! Figure 11-1 shows one such plausible attack scenario.

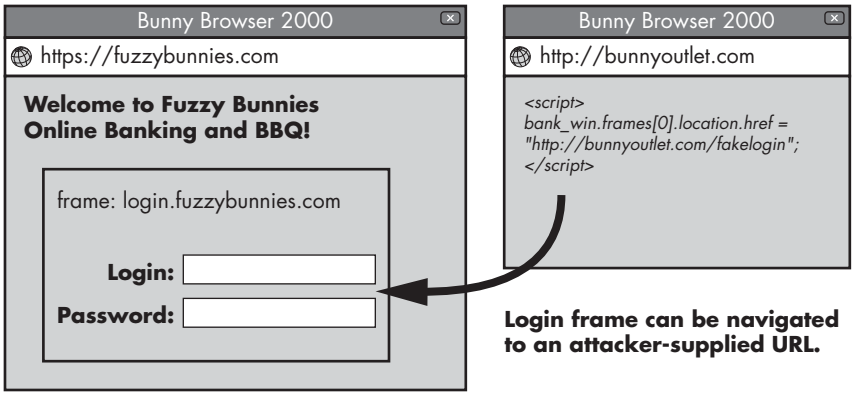


Figure 11-1: A historically permitted, dangerous frame navigation scenario: The window on the right is opened at the same time as a banking website and is actively subverting it.

Georgi Guninski, one of the pioneering browser security researchers, realized as early as 1999 that by permitting unconstrained frame navigation, we were headed for some serious trouble. Following his reports, vendors attempted to roll out frame navigation restrictions mid-2000.[1] Their implementation constrained all cross-frame navigation to the scope of a single window, preventing malicious web pages from interfering with any other simultaneously opened browser sessions.

Surprisingly, even this simple policy proved difficult to implement correctly. It was only in 2008 that Firefox eliminated this class of problems,[2] while Microsoft essentially ignored the problem until 2006. Still, these setbacks aside, we should be fine—right?

Frame Descendant Policy and Cross-Domain Communications

The simple security restriction discussed in the previous session was not, in fact, enough. The reason was a new class of web applications, sometimes known as *mashups*, that combined data from various sources to enable users to personalize their working environment and process data in innovative ways. Unfortunately for browser vendors, such web applications frequently relied on third-party gadgets loaded through *<iframe>* tags, and their developers

* One potential attack is this: Open a legitimate website (say, *http://trusted-bank.com/*) in a new window, wait for the user to inspect the address bar, and then quickly change the location to an attacker-controlled but similarly named site (e.g., *http://trustea-bank.com/*). The likelihood of successfully phishing the victim may be higher than when the user is navigating to the bad URL right away.

could not reasonably expect that loading a single frame from a rogue source would put all other frames on the page at risk. Yet, the simple and elegant window-level navigation policy amounted to permitting exactly that.

Around 2006, Microsoft agreed that the current approach was not sustainable and developed a more secure *descendant policy* for frame navigation in Internet Explorer 7. Under this policy, navigation of non-same-origin frames is permitted only if the party requesting the navigation shares the origin with one of the ancestors of the targeted view. Figure 11-2 shows the navigation scenario permitted by this new policy.

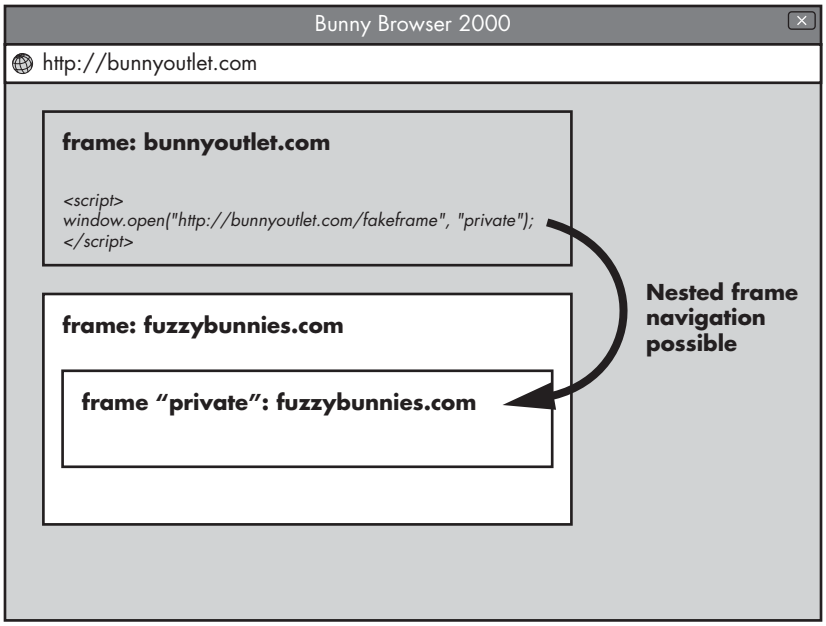


Figure 11-2: A complex but permissible navigation between non-same-origin frames. This attempt succeeds only because the originating frame has the same origin as one of the ancestors of the targeted document—here, it's the top-level page itself.

As with many other security improvements, Microsoft never backported this policy to the still popular Internet Explorer 6, and it never convincingly pressured users to abandon the older and increasingly insecure (but still superficially supported) version of its browser. On a more positive note, by 2009, three security researchers (Adam Barth, Collin Jackson, and John C. Mitchell) convinced Mozilla, Opera, and WebKit to roll out a similar policy in their browsers,[3] finally closing the mashup loophole for a good majority of the users of the Internet.

Well, *almost* closing it. Even the new, robust policy has a subtle flaw. Notice in Figure 11-2 that a rogue site, *http://bunnyoutlet.com/*, can interfere with a private frame that *http://fuzzybunnies.com/* has created for its own use. At first glance, there is no harm here: The attacker's domain is shown in the address bar, so the victim, in theory, should not be fooled into interacting with the subverted UI of *http://fuzzybunnies.com/* in any meaningful way. Sadly, there is a catch: Some web applications have learned to use frames not to

create user interfaces but to relay programmatic messages between origins. For applications that need to support Internet Explorer 6 and 7, where *postMessage(…)* is not available, the tricks similar to the approach shown in Figure 11-3 are commonplace.

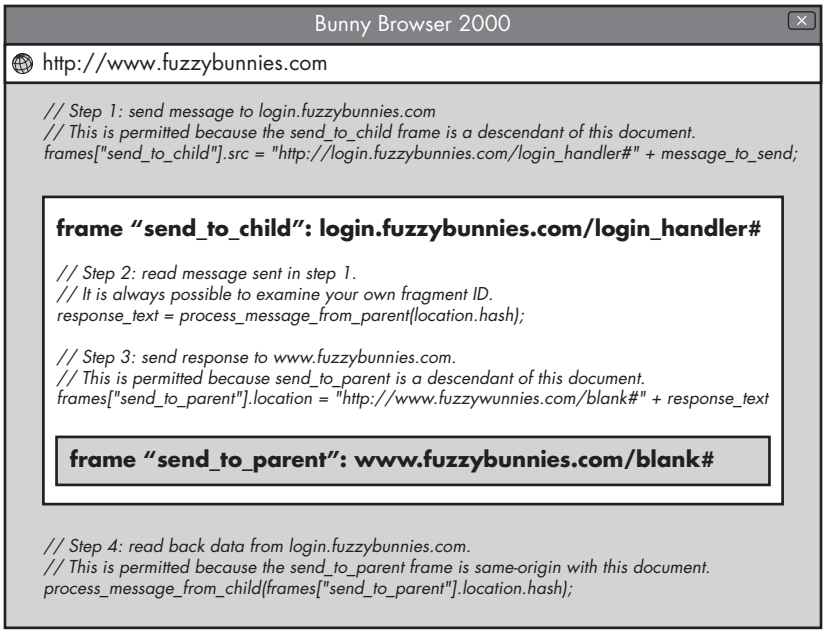


Figure 11-3: A potential cross-domain communication scheme, where the top-level page encodes messages addressed to the embedded gadget in the fragment identifier of the gadget frame and the gadget responds by navigating a subframe that is same-origin with the top-level document. If this application is framed on a rogue site, the top-level document controlled by the attacker will be able to inject messages between the two parties by freely navigating send_to_parent *and* send_to_child.

If an application that relies on a similar hack is embedded by a rogue site, the integrity of the communication frames may be compromised, and the attacker will be able to inject messages into the stream. Even the uses of *postMessage(…)* may be at risk: If the party sending the message does not specify a destination origin or if the recipient does not examine the originating location, hijacking a frame will benefit the attacker in exactly the same way.

Unsolicited Framing

The previous discussion of cross-frame navigation highlights one of the more interesting weaknesses in the browser security model, as well as the disconnect between the design goals of HTML and the aim of the same-origin policy. But that's not all: The concept of cross-domain framing is, by itself, fairly risky. Why? Well, any malicious page may embed a third-party application without a user's knowledge, let alone consent. Further, it may obfuscate this fact by overlaying other visual elements on top of the frame, leaving visible just a small chunk of the original site, such as a button that performs a state-changing

action. In such a setting, any user logged into the targeted application with ambient credentials may be easily tricked into interacting with the disguised UI control and performing an undesirable and unintended action, such as changing sharing settings for a social network profile or deleting data.

This attack can be improved by the rogue site leveraging a CSS2 property called *opacity* to make the targeted frame completely invisible without affecting its actual behavior. Any click in the area occupied by such a see-through frame will be delivered to the UI controls contained therein (see Figure 11-4). Too, by combining CSS opacity with JavaScript code to make the frame follow the mouse pointer, it is possible to carry out the attack fairly reliably in almost any setting: Convincing the user to click anywhere in the document window is not particularly hard.

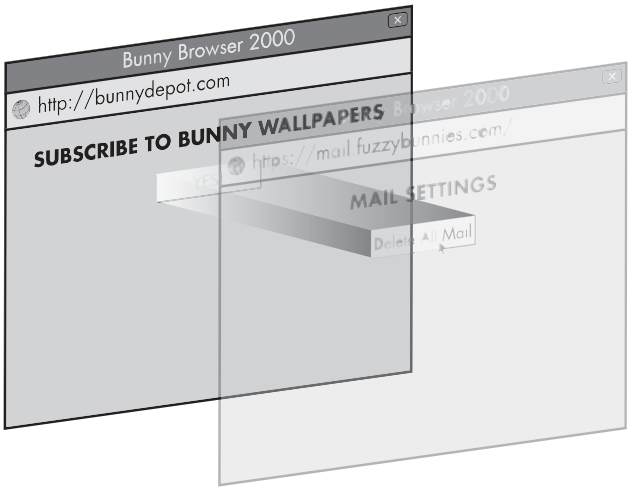


Figure 11-4: A simplified example of a UI-splicing attack that uses CSS opacity to hide the document the user will actually interact with

Researchers have recognized the possibility of such trickery to some extent since the early 2000s, but a sufficiently convincing attack wasn't demonstrated until 2008, when Robert Hansen and Jeremiah Grossman publicized the issue broadly.[4] Thus, the term *clickjacking* was born.

The high profile of Hansen and Grossman's report, and their interesting proof-of-concept example, piqued vendors' interest. This interest proved to be short-lived, however, and there appears to be no easy way to solve this problem without taking some serious risks. The only even remotely plausible way to mitigate the impact would be to add renderer-level heuristics to disallow event delivery to cross-domain frames that are partly obstructed or that have not been displayed long enough. But this solution is complicated and hairy enough to be unpopular.[5] Instead, the problem has been slapped with a Band-Aid. A new HTTP header, *X-Frame-Options*, permits concerned sites to opt out of being framed altogether (*X-Frame-Options: deny*) or consent only to framing within a single origin (*X-Frame-Options: same-origin*).[6] This header

is supported in all modern browsers (in Internet Explorer, beginning with version 8),* but it actually does little to address the vulnerability.

Firstly, the opt-in nature of the defense means that most websites will not adopt it or will not adopt it soon enough; in fact, a 2011 survey of the top 10,000 destinations on the Internet found that barely 0.5 percent used this feature.[7]

To add insult to injury, the proposed mechanism is useless for applications that want to be embedded on third-party sites but that wish to preserve the integrity of their UIs. Various mashups and gadgets, those syndicated "like" buttons provided by social networking sites, and managed online discussion interfaces are all at risk.

Beyond the Threat of a Single Click

As the name implies, the clickjacking attack outlined by Grossman and Hansen targets simple, single-click UI actions. In reality, however, the problem with deceptive framing is more complicated than the early reporting would imply. One example of a more complex interaction is the act of selecting, dragging, and dropping a snippet of text. In 2010, Paul Stone proposed a number of ways in which such an action could be disguised as a plausible interaction with an attacker's site,[8] the most notable of which is the similarity between drag-and-drop and the use of a humble document-level scrollbar. The same click-drag-release action may be used to interact with a legitimate UI control or to unwittingly drag a portion of preselected text out of a sensitive document and drop it into an attacker-controlled frame. (Cross-domain drag-and-drop is no longer permitted in WebKit, but as of this writing other browser vendors are still debating the right way to address this risk.)

An even more challenging problem is keystroke redirection. Sometime in 2010, I noticed that it was possible to selectively redirect keystrokes across domains by examining the code of a pressed key using the *onkeydown* event in JavaScript. If the pressed key matched what a rogue site wanted to enter into a targeted application, HTML element focus could be changed momentarily to a hidden <*iframe*>, thereby ensuring the delivery of the actual keystrokes to the targeted web application rather than the harmless text field the user seems to be interacting with.[9] Using this method, an attacker can synthesize arbitrarily complex text in another domain on the user's behalf—for example, inviting the attacker as an administrator of the victim's blog.

Browser vendors addressed the selective keystroke redirection issue by disallowing element focus changes in the middle of a keypress, but doing so did not close the loophole completely. After all, in some cases, an attacker can predict what key will be pressed next and roughly at what time, thereby permitting a preemptive, blindly executed focus switch. The two most obvious cases are a web-based action game or a typing-speed test, since both typically involve rapid pressing of attacker-influenced keys.

* In older versions of Internet Explorer, web application developers sometimes resort to JavaScript in an attempt to determine whether the *window* object is the same as *parent*, a condition that should be satisfied if no higher-level frame is present. Unfortunately, due to the flexibility of JavaScript DOM, such checks, as well as many types of possible corrective actions, are notoriously unreliable.

In fact, it gets better: Even if a malicious application only relies on free-form text entry—for example, by offering the user a comment-submission form—it's often possible to guess which character will be pressed next based on the previous few keystrokes alone. English text (and text in most other human languages) is highly redundant, and in many cases, a considerable amount of input can be predicted ahead of time: You can bet that *a-a-r-d-v* will be followed by *a-r-k*, and almost always you will be right.

Cross-Domain Content Inclusion

Framing and navigation are a distinct source of trouble, but these mechanisms aside, HTML supports a number of other ways to interact with non-same-origin data. The usual design pattern for these features is simple and seemingly safe: A constrained data format that will affect the appearance of the document is retrieved and parsed without being directly shown to the origin that referenced it. Examples of mechanisms that follow this rule include markup such as *<script src=…>*, *<link rel=stylesheet href=…>*, *<img src=…>*, and several related cases discussed throughout Part I of this book.

Regrettably, the devil is in the details. When these mechanisms were first proposed, nobody asked several extremely pressing questions:

- Should these subresources be requested with ambient credentials associated with their origin? If so, there is a danger that the response may contain sensitive data not intended for the requesting party. It would probably be better to require some explicit form of authentication or to notify the server about the origin of the requesting page.
- Should the relevant parsers be designed to minimize the risk of mistaking one document type for another? And should the servers have control over how their responses are interpreted (for example through the *Content-Type* header)? If not, what are the consequences of, say, interpreting a user's private JPEG image as a script?
- Should the requesting page have no way to infer anything about the contents of the retrieved payloads? If yes, then this goal needs to be taken into account with utmost care when designing all the associated APIs. (If such separation is not a goal, the importance of the previous questions is even more pronounced.)

The developers acted with conflicting assumptions about these topics, or perhaps had not given them any thought at all, leading to a number of profound security risks. For example, in most browsers, it used to be possible to read arbitrary, cookie-authenticated text by registering an *onerror* handler on cross-domain *<script>* loads: The verbose "syntax error" message generated by the browser would include a snippet of the retrieved file. Still, no problem in this category is more interesting than a glitch discovered by Chris Evans in 2009.[10] He noticed that the hallmark fault tolerance of CSS parsers (which, as you may recall, recover from syntax errors by attempting to resynchronize at the nearest curly bracket) is also a fatal security flaw.

In order to understand the issue, consider the following simple HTML document. This document contains two occurrences of an attacker-controlled

string, and—sandwiched in between—a sensitive, user-specific value (in this case, a user's name):

```
<head>
  <title>Page not found: ');} gotcha { background-image: url('/</title>
</head>
<body>
  ...
  <span class="header">You are logged in as: John Doe</span>
  ...
  <div class="error_message">
  Page not found: ');} gotcha { background-image: url('/
  </div>
  ...
</body>
```

Let's assume that the attacker lured the victim to his own page and, on this page, used *<link rel=stylesheet>* to load the aforementioned cross-domain HTML document in place of a stylesheet. The victim's browser will happily comply: It will request the document using the victim's cookies, will ignore *Content-Type* on the subsequent response, and will hand the retrieved content over to the CSS parser. The parser will cheerfully ignore all syntax errors leading up to what appears to be a CSS rule named *gotcha*. It will then process the *url('...* pseudo-function, consuming all subsequent HTML (including the secret user name!), until it reaches a matching quote and a closing parenthesis. When this faux stylesheet is later applied to a *class=gotcha* element on the attacker's website, the browser will attempt to load the resulting URL and will leak the secret value to the attacker's server in the process.

Astute readers may note that the CSS standard does not support multiline string literals, and as such, this trick would not work as specified. That's partly true: In most browsers, the attempt will succeed only if the critical segment of the page contains no stray newlines. Some web applications are optimized to avoid unnecessary whitespaces and therefore will be vulnerable, but most web developers use newlines liberally, thwarting the attack. Alas, as noted in Chapter 5, one browser behaves differently: Internet Explorer accepts multiline strings in stylesheets and many other egregious syntax violations, accidentally amplifying the impact of this flaw.

NOTE *Since identifying this problem, Chris Evans has pushed for fixes in all mainstream browsers, and as of this writing, most implementations reject cross-domain stylesheets that don't begin right away with a valid CSS rule or that are served with an incompatible* Content-Type *header (same-origin stylesheets are treated less restrictively). The only vendor to resist was Microsoft, which changed its mind only after a demonstration of a successful proof-of-concept attack against Twitter.*[11] *Following this revelation, Microsoft agreed not only to address the problem in Internet Explorer 8 but also—uncharacteristically—to backport this particular fix to Internet Explorer 6 and 7 as well.*

Thanks to Chris's efforts, stylesheets are a solved problem, but similar problems are bound to recur for other types of cross-domain subresources. In such cases, not all transgressions can be blamed on the sins of the old. For

example, when browser vendors rolled out *<canvas>*, a simple HTML5 mechanism that enables JavaScript to create vector and bitmap graphics,[12] many implementations put no restrictions on loading cross-domain images onto the canvas and then reading them back pixel by pixel. As of this writing, this issue, too, has been resolved: A canvas once touched by a cross-domain image becomes "tainted" and can only be written to, not read. But when we need to fix each such case individually, something is very wrong.

A Note on Cross-Origin Subresources

So far, we have focused on the risks of malicious websites navigating or including content that belongs to trusted parties. That said, the ability to load certain types of subresources from other origins has significant consequences, even if not actively subverted by a third-party site.

In Part I of the book, we hinted that loading a script or a stylesheet from another origin effectively equates the security of the document that performs the load to the security of the origin of the loaded subresource; in particular, loading an HTTP script on an HTTPS page undoes most of the benefits of encryption. Similarly, loading a script from a provider whose infrastructure is vulnerable to attack can be nearly as problematic as not properly maintaining your own servers.

In addition to scripts and stylesheets, other content types that may lead to serious trouble include remote fonts (a recent addition to CSS) and plug-ins with access to the embedding page (such as *allowScriptAccess=always* for Flash). It is also somewhat dangerous to load images, icons, cursors, or HTML frames from untrusted sources, although the impact of doing so is contained to some extent and will be use specific.

Contemporary browsers attempt to detect cases where HTTPS documents load HTTP resources—a condition known as *mixed content*. They do so fairly inconsistently, however: Internet Explorer is the only browser that blocks most types of mixed content by default (and Chrome is expected to follow suit), but neither Internet Explorer nor Firefox nor Opera consistently detects mixed content on *<embed>*, *<object>*, or *<applet>* tags. In browsers other than Internet Explorer, the default action is a subtle warning (for example, an exclamation mark next to the lock icon) or a cryptic dialog, which does very little to protect the user but which may alert a sufficiently attentive web developer.

As to the other flavor of mixed content—loading subresources across domains that offer different levels of trust—browsers have no way to detect this. The decision to include content from dubious sources is often made too lightly and such mistakes can be difficult to spot until too late.

NOTE *Another interesting problem with cross-domain subresources is that they may request certain additional permissions or credentials from the browser. The associated browser security prompts are usually not designed with such scenarios with mind, and they do not always make sufficiently clear which origin is requesting the permission and based on what sort of relationship with the top-level site. We discussed one such problem in Chapter 3: the authentication prompt shown in response to HTTP code 401. Several other, related cases will appear in Chapter 15.*

Privacy-Related Side Channels

Another unfortunate and noteworthy consequence of the gaps in the same-origin policy is the ability to collect information about a user's interaction with unrelated sites. Some of the most rudimentary examples, most of them known for well over a decade,[13] include the following:

- Using *onload* handlers to measure the time it takes to load certain documents, an indication of whether they have been previously visited and cached by the browser or not.[14]
- Using *onload* and *onerror* on <*img*> tags to see if an authentication-requiring image on a third-party site can be loaded, thus disclosing whether the user is logged into that site or not. (Bonus: Sometimes, the error message disclosed to the *onerror* handler will include snippets of the targeted page, too.)
- Loading an unrelated web application in a hidden frame and examining properties such as the number and names of subframes created on that page (available through the <*handle*>*.frames[]* array) or the set of global variables (sometimes leaked through the semantics of the *delete* operator) in order to detect the same. Naturally, the set of sites the user visits or is logged into can be fairly sensitive.

In addition to these tricks, a particularly frightening class of privacy problems is associated with two APIs created several years ago to help websites understand the style applied to any document element (the sum of browser-specific defaults, CSS rules, and any runtime tweaks made automatically by the browser or performed via JavaScript). The two APIs in question are *getComputedStyle*, mandated by CSS Level 2,[15] and *currentStyle*, proprietary to Internet Explorer.[16] Their functionality, together with the ability to assign distinctive styling to visited links (using the *:visited* pseudo-class), means that any rogue JavaScript can rapidly display and examine thousands of URLs to see which ones are shaded differently (due to being present in a user's browsing history), thereby building a reliable, extensive, and possibly incriminating overview of a user's online habits with unprecedented efficiency and reliability.

This problem has been known since at least since 2002, when Andrew Clover posted a brief note about it to the popular BUGTRAQ mailing list.[17] The issue received little scrutiny in the following years, until a series of layperson-targeted demonstrations and a subsequent public outcry around 2006. A few years later, Firefox and WebKit browsers rolled out security improvements to limit the extent of styling possible in *:visited* selectors and to limit the ability to inspect the resulting composite CSS data.

That said, such fixes will never be perfect. Even though they make automated data collection impossible, smaller quantities of data can be obtained with a user's help. Case in point: Collin Jackson and several other researchers proposed a simple scheme that involved presenting a faux

CAPTCHA* consisting of seven-segment, LCD-like digits.[18] Rather than being an actual, working challenge, the number the user would see depended on the *:visited*-based styling applied to superimposed links (see Figure 11-5); by typing that number back onto the page, the user would unwittingly tell the author of the site what exact styling had been applied and, therefore, what sites appeared in the victim's browsing history.

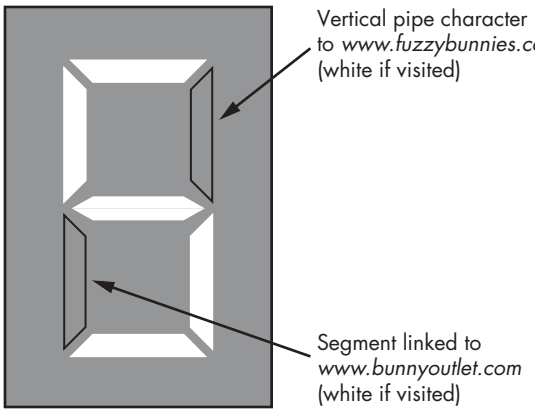


Figure 11-5: A fake seven-segment display can be used to read back link styling when the displayed number is entered into the browser in an attempt to solve a CAPTCHA. The user will see 5, 6, 9, or 8, depending on prior browsing history.

Other SOP Loopholes and Their Uses

Although this chapter has focused on areas where the limitations of the same-origin policy have a clear, negative impact on the security or privacy of online browsing, there are several accidental gaps in the scheme that in most cases seem to be of no special consequence. For example, in many versions of Internet Explorer, it was possible to manipulate the value of *window.opener* or *window.name* of an unrelated window. Meanwhile in Firefox, there are currently no constraints on setting *location.hash* across domains, even though all other partial location properties are restricted.

The primary significance of these mechanisms is that they are often repurposed to build cross-domain communication channels in browsers that do not support the *postMessage(...)* API. Such mechanisms are often built on shaky ground: The lack of SOP enforcement is typically uniform and means that any website, not just the "authorized" parties, will be able to interfere with the data. The ability for rogue parties to navigate nested frames, as discussed in "Frame Hijacking Risks" on page 175, further complicates the picture.

* CAPTCHA (sometimes expanded as Completely Automated Public Turing test to tell Computers and Humans Apart) is a term for a security challenge that is believed to be difficult to solve using computer algorithms but that should be easy for a human being. It is usually implemented by showing an image of several randomly selected, heavily distorted characters and asking the user to type them back. CAPTCHA may be used to discourage the automation of certain tasks, such as opening new accounts or sending significant volumes of email. (Needless to say, due to advances in computer image processing, robust CAPTCHAs are increasingly difficult for humans to solve, too.)

Security Engineering Cheat Sheet

Good Security Hygiene for All Websites

- ☑ Serve all content for your site with *X-Frame-Options: sameorigin*. Make case-by-case exceptions only for specific, well-understood locations that require cross-domain embedding. Try not to depend on JavaScript "framebusting" code to prevent framing because it's very tricky to get that code right.
- ☑ Return user-specific, sensitive data that is not meant to be loaded across domains using well-constrained formats that are unlikely to be misinterpreted as standalone scripts, stylesheets, and so on. Always use the right *Content-Type*.

When Including Cross-Domain Resources

- ☑ In many scenarios (especially when dealing with scripts, stylesheets, fonts, and certain types of plug-in-handled content), you are linking the security of your site to the originating domain of the subresource. When in doubt, make a local copy of the data instead. On HTTPS sites, require all subresources to be served over HTTPS.

When Arranging Cross-Domain Communications in JavaScript

- ☑ Consult the cheat sheet in Chapter 9. Do not use cross-frame communication schemes based on *location.hash*, *window.name*, *frameElements*, and similar ephemeral hacks, unless you are prepared to deal with injected content.
- ☑ Do not expect subframes on your page to sit still, especially if you are not using *X-Frame-Options* to limit the ability of other sites to frame your application. In certain cases, an attacker may be able to navigate such frames to a different location without your knowledge or consent.

12

OTHER SECURITY BOUNDARIES

All previously described origin-level content-isolation policies, and the accompanying context inheritance and document navigation logic, work hand in hand to form the bulk of the browser security model. Impenetrable and fragile, that model is also incomplete: A handful of interesting corner cases completely escape any origin-based frameworks.

The security risks associated with these corner cases can't be addressed simply by fine-tuning the mechanisms discussed earlier in this book. Instead, additional, sometimes hopelessly imperfect security boundaries need to be created from scratch. These new boundaries may, for example, further restrict the ability of rogue web pages to navigate to certain URLs.

This chapter offers a quick look at some of the most significant examples of the loopholes in the origin-based model and the ways that vendors have dealt with them.

Navigation to Sensitive Schemes

In the past, browser vendors reasoned that there was no harm in allowing any page on the Internet to navigate to a document stored on a user's hard drive using the *file:* protocol or to open a new window pointing to a privileged resource, such as the *about:config* page in Firefox. After all, they thought, the originating document and the destination would not be same-origin, and, therefore, any direct access to the sensitive data would be prevented.

For many years, based on this rationale, browsers permitted such navigation to take place. Alas, this decision proved to be not only extremely confusing[*] but also dangerous. The danger comes from the fact that many programs, browsers included, tend to store various types of Internet-originating content in the filesystem; temporary files and cached documents are a common example. In many cases, an attacker could have some control over the creation and contents of such files, and, if the resources are created at a predictable location, subsequent navigation to the right *file:* URL could allow the attacker to execute his own payload in this coveted origin, with access to any other file on the disk and, perhaps, any other website on the Internet.

Comparably disastrous consequences have been observed with a variety of privileged, internally handled URLs. The ability to navigate directly to locations such as *about:config* (Firefox) not only made it possible to exploit potential vulnerabilities in the privileged scripts (a transgression to which browser vendors are not immune) but also led to system compromise if, through a literal application of the same-origin policy, the browser naïvely deemed *about:config* and *about:blank* to come from the same origin.

Having learned from a history of painful mishaps, modern browsers typically police navigation based on three tiers of URL schemes:

- **Unrestricted** This category includes virtually all true network protocols, such as HTTP, HTTPS, FTP; most encapsulating pseudo-protocols such as *mhtml:* or *jar:*; and all schemes registered to plug-ins and external applications. Navigation to these URLs is not constrained in any specific way.
- **Partly restricted** This category includes several security-sensitive schemes such as *file:* and special pseudo-URLs such as *javascript:* or *vbscript:*. Navigation to them is not completely denied, but it is subject to additional, scheme-specific security checks. For example, access to *file:* is usually permitted only from other *file:* documents, requiring the first one to be opened manually. (The rules for navigation to *javascript:* URLs were discussed in Chapter 10.)
- **Fully restricted** This category includes privileged pages in *about:*, *res:*, *chrome:*, and similar browser-specific namespaces. Normal, unprivileged HTML documents are not permitted to navigate to them under any circumstance.

[*] For example, on Windows systems, a common prank was to use a seamlessly embedded *<iframe>* pointing to *file:///c:/* in order to display the contents of a victim's hard drive, leading some users to believe that the page doing so has somehow gained access to their files.

Access to Internal Networks

The trouble with accessing sensitive protocols is merely a prelude to a far more serious issue that somehow escaped the creators of the same-origin policy. The problem is that DNS-derived origins may have nothing to do with actual network-level boundaries—or with how these boundaries change over time. A malicious script may be granted same-origin access to intranet sites on the victim's local network, even if a firewall prevents the attacker from interacting with these destinations directly.

There are at least three distinctive venues for such attacks.

Origin Infiltration

When a user visits a rogue network—such as an open wireless network at an airport or in a café—an attacker on that network may trick the victim's browser into opening a URL such as *http://us-payroll/*. When this happens, the attacker may provide his own, spoofed content for that site. Frighteningly, if the user then brings the same browser to a corporate network, the previously injected content will have same-origin access to the real version of *http://us-payroll/*, complete with the user's ambient credentials.

The persistence of injected content may be achieved in a couple of ways. The most basic method is for an attacker simply to inject a hidden *http://us-payroll/* frame onto every visited page in the hope that the user will suspend a portable computer with the browser still running and then take it to another network. Another technique is *cache poisoning*: creating long-lived, cached objects that the browser will use instead of retrieving a fresh copy from the destination site. Several other, more obscure approaches also exist.

DNS Rebinding

This arguably less serious but more easily exploitable problem was mentioned in footnote 1 in Chapter 9. In short, since the same-origin policy looks just at the DNS name of a host, not at the IP address, an attacker who owns *bunnyoutlet.com* is free to respond initially to a DNS lookup from a user with a public IP such as *213.134.128.25* and then switch to an address reserved for private networks, such as *10.0.0.1*. Documents loaded from both sources will be considered same-origin, giving the attacker the ability to interact with the victim's internal network.

The mitigating factor is that this interaction will not involve proper ambient credentials that the victim normally has for the targeted site: As far as the browser is concerned, it is still talking to *bunnyoutlet.com* and not to, say, the aforementioned *us-payroll* site. Still, the prospect of the attacker examining the internal network and perhaps trying to brute-force the appropriate credentials or identify vulnerabilities is disconcerting.

Simple Exploitation of XSS or XSRF Flaws

Even outside the realm of the same-origin policy, the mere possibility of navigating to intranet URLs means that the attacker may attempt to (blindly) target known or suspected vulnerabilities in locally running software. Because internal applications are thought to be protected from malicious users, they are often not engineered or maintained to the same standards as externally facing code.

One striking example of this problem is the dozens of vulnerabilities discovered over the years in internal-only web management interfaces of home network routers manufactured by companies such as Linksys (Cisco), Netgear, D-Link, Motorola, and Siemens. Cross-site request forgery vulnerabilities in these applications can, in extreme cases, permit attackers to access the device and intercept or modify all network traffic going to or through it.

So far, the disconnect between browser security mechanisms and network segmentation remains an unsolved problem in browser engineering. Several browsers try to limit the impact of DNS rebinding by caching DNS responses for a predefined time—a practice known as *DNS pinning*—but the defense is imperfect, and the remaining attack vectors still remain.

NOTE *Unusually, Internet Explorer takes the lead on this front, offering an optional way to mitigate the risk. Microsoft's users are protected to some extent if they flip a cryptic zone setting named "websites in less privileged web content zone can navigate into this zone" to "disable" in the configuration options for local intranet. Unfortunately, the zone model in Internet Explorer comes with some unexpected pitfalls, as we'll discuss in Chapter 15.*

Prohibited Ports

Security researchers have cautioned that the ability of browsers to submit largely unconstrained cross-origin request bodies, for example with *<form method="POST" enctype="text/plain">*, may interfere with certain other fault-tolerant but non-HTTP network services. For example, consider SMTP, the dominant mail transfer protocol: When interacting with an unsuspecting browser, most servers that speak SMTP will patiently ignore the first few incomprehensible lines associated with HTTP headers and then honor any SMTP commands that appear in the request body. In effect, the browser could be used as a proxy for relaying spam.

A related but less well-explored concern, discussed in Chapter 3, is the risk of an attacker talking to non-HTTP services running in the same domain as the targeted web application and tricking the browser into misinterpreting the returned, possibly partly attacker-controlled data as HTML delivered over HTTP/0.9. This behavior could expose cookies or other credentials associated with the targeted site.

The design of HTTP makes it impossible to solve these problems in a particularly robust way. Instead, browser vendors have responded in a rather unconvincing manner: by shipping a list of prohibited TCP ports

to which requests cannot be sent. For Internet Explorer versions 6 and 7, the list consists of the following port numbers:

19	chargen
21	ftp
25	smtp
110	pop3
119	nntp
143	imap2

Versions 8 and 9 of Internet Explorer further prohibit ports 220 (imap3) and 993 (ssl imap3).

All other browsers discussed in this book use a different, common list:

1	tcpmux	115	sftp
7	echo	117	uccp-path
9	discard	119	nntp
11	systat	123	ntp
13	daytime	135	loc-srv
15	netstat	139	netbios
17	qotd	143	imap2
19	chargen	179	bgp
20	ftp-data	389	ldap
21	ftp	465	ssl smtp
22	ssh	512	exec
23	telnet	513	login
25	smtp	514	shell
37	time	515	printer
42	name	526	tempo
43	nicname	530	courier
53	domain	531	chat
77	priv-rjs	532	netnews
79	finger	540	uucp
87	ttylink	556	remotefs
95	supdup	563	ssl nntp
101	hostriame	587	smtp submission
102	iso-tsap	601	syslog
103	gppitnp	636	ssl ldap
104	acr-nema	993	ssl imap
109	pop2	995	ssl pop3
110	pop3	2049	nfs
111	sunrpc	4045	lockd
113	auth	6000	X11

There are, of course, various protocol-specific exceptions to these rules. For example, *ftp:* URLs are obviously permitted to access port 21, normally associated with that protocol.

The current solution is flawed in several ways, the most important of which may be that both lists have numerous glaring omissions and, given the number of network protocols devised to date, simply have no chance of ever being exhaustive. For example, no rule would prevent the browser from talking to Internet Relay Chat (IRC) servers, which use a fault-tolerant, text-based protocol not entirely unlike SMTP.

The lists are also not regularly updated to reflect the demise of nearly extinct network protocols or the introduction of new ones. Lastly, they can unfairly and unexpectedly penalize system administrators for picking non-standard ports for certain services they want to hide from public view: Doing so means opting out of this browser-level protection mechanism.

Limitations on Third-Party Cookies

Since their inception, HTTP cookies have been misunderstood as the tool that enabled online advertisers to violate users' privacy to an unprecedented and previously unattainable extent. This sentiment has been echoed by the mainstream press in the years since. For example, in 2001, the *New York Times* published a lengthy exposé on the allegedly unique risks of HTTP cookies and even quoted Lawrence Lessig, a noted legal expert and a political activist:[1]

> Before cookies, the Web was essentially private. After cookies, the Web becomes a space capable of extraordinary monitoring.

The high-profile assault on a single HTTP header continued over the course of a decade, gradually shifting its focus toward third-party cookies in particular. Third-party cookies are the cookies set by domains other than the domain of the top-level document, and they are usually associated with the process of loading images, frames, or applets from third-party sites. The reason they have attracted attention is that operators of advertising networks have embraced such cookies as a convenient way to tag a user who sees their ad embedded on *fuzzybunnies.com* and then recognize that user through a similar embedded ad served on *playboy.com*.

Because the clearly undesirable possibility of performing this type of cross-domain tracking has been erroneously conflated with the existence of third-party cookies, the pressure on browser vendors has continued to mount. In one instance, the *Wall Street Journal* flat out accused Microsoft of being in bed with advertisers for not eliminating third-party cookies in the company's product.[2]

Naturally, the readers of this book will recognize that the fixation on HTTP cookies is deeply misguided. There is no doubt that some parties use the mechanism for vaguely sinister purposes, but nothing makes it uniquely suited for this task; there are many other equivalent ways to store unique identifiers on visitors' computers (such as cache-based tags, previously discussed in Chapter 3). Besides, it is simply impossible to prevent cooperating sites

from using existing unique fingerprints of every browser (exposed through the JavaScript object model or plug-ins such as Flash) to correlate and mine cross-domain browsing patterns at will. The sites that embed advertisements for profit are quite willing to cooperate with the parties who pay their bills.

In fact, the common reliance on HTTP cookies offers a distinctive advantage to users: Unlike many of the easily embraced alternatives, this mechanism is purpose built and coupled with reasonably well-designed and fine-grained privacy controls. Breaking cookies will not hinder tracking but will remove any pretense of transparency from the end user. Another noted privacy and security activist, Ed Felten, once said: "If you're going to track me, please use cookies."[3]

Unscrupulous online tracking is a significant social issue, and new technical mechanisms may be needed so that users can communicate their privacy preferences to well-behaved sites (such as the recently added *DNT* request header[4] rolled out in Firefox 4). In order to deal with the ill-behaved ones, a regulatory framework may be required, too. In the absence of such a framework, in Internet Explorer 9, Microsoft is experimenting with a managed blacklist of known bad sources of tracking cookies—but the odds that this would discourage sleazy business practices are slim.

In any case, despite having little or no merit, the continued public outcry against third-party cookies eventually resulted in several browser vendors shipping half-baked and easily circumvented solutions that let them claim they had done *something*.

- In Internet Explorer, setting and reading third-party cookies is blocked by default, except for session cookies accompanied by a satisfactory P3P header. *P3P* (*Platform for Privacy Preferences*)[5] is a method to construct machine-readable, legally binding summaries of a site's privacy policy, be it as an XML file or as a *compact policy* in an HTTP header. For example, the keyword TEL in an HTTP header means that the site uses the collected information for telemarketing purposes. (No technical measure will prevent a site from lying in a P3P header, but the potential legal consequences are meant to discourage that.)

  NOTE *The incredibly ambitious, 111-page P3P specification caused the solution to crumble under its own weight. Large businesses are usually very hesitant to embrace P3P as a solution to technical problems because of the legal footprint of the spec, while small businesses and individual site owners copy over P3P header recipes with little or no understanding of what they are supposed to convey.*

- In Safari, the task of setting third-party cookies is blocked by default, but previously issued cookies can be read freely. However, this behavior can be overridden if the user interacts with the cookie-setting document first. Such an interaction could be intentional but may very well not be: The clickjacking-related tricks outlined in Chapter 11 apply to this scenario as well.

- In other browsers, third-party cookies are permitted by default, but a configuration option is provided to change the behavior. Enabling this option limits the ability to set third-party cookies, but reading existing ones is not limited in any way.

For the purpose of these checks, a cookie is considered to be coming from a third party if it's loaded from a completely unrelated domain. For example, a frame pointing to *bunnyoutlet.com* loaded on *fuzzybunnies.com* meets this criterion, but *www1.fuzzybunnies.com* and *www2.fuzzybunnies.com* are considered to be in a first-party relationship. The logic used to make this determination is fragile, and it suffers from the same problems that cookie *domain* scoping would. In Internet Explorer 6 and 7, for example, the comparisons in certain country-level domains are performed incorrectly.

NOTE *The crusade against third-party cookies could be seen as a harmless exercise, but it has had negative consequences, too. Browsers that reject third-party cookies make it very difficult to build cookie-based authentication for embeddable gadgets and other types of mashups, and they make it difficult to use "sandbox" domains to isolate untrusted but private content from the main application to limit the impact of script-injection flaws.*

Security Engineering Cheat Sheet

When Building Web Applications on Internal Networks

☑ Assume that determined attackers will be able to interact with those applications through a victim's browser, regardless of any network-level security controls. Ensure that proper engineering standards are met and require HTTPS with *secure* cookies for all sensitive applications in order to minimize the risk of origin infiltration attacks.

When Launching Non-HTTP Services, Particularly on Nonstandard Ports

☑ Evaluate the impact of browsers unintentionally issuing HTTP requests to the service and the impact of having the response interpreted as HTTP/0.9. For vulnerable protocols, consider dropping the connection immediately if the received data begins with "GET" or "POST" as one possible precaution.

When Using Third-Party Cookies for Gadgets or Sandboxed Content

☑ If you need to support Internet Explorer, be prepared to use P3P policies (and evaluate their legal significance). If you need to support Safari, you may have to resort to an alternative credential storage mechanism (such as HTML5 *localStorage*).

13

CONTENT RECOGNITION MECHANISMS

So far, we have looked at a fair number of well-intentioned browser features that, as the technology matured, proved to be short-sighted and outright dangerous. But now, brace for something special: In the history of the Web, nothing has proven to be as misguided as *content sniffing*.

The original premise behind content sniffing was simple: Browser vendors assumed that in some cases, it would be appropriate—even desirable—to ignore the normally authoritative metadata received from the server, such as the *Content-Type* header. Instead of honoring the developer's declared intent, implementations that support content sniffing may attempt to second-guess the appropriate course of action by applying proprietary heuristics to the returned payload in order to compensate for possible mistakes. (Recall from Chapter 1 that during the First Browser Wars, vendors turned fault-tolerance compatibility into an ill-conceived competitive advantage.)

It didn't take long for content-sniffing features to emerge as a substantial and detrimental aspect of the overall browser security landscape. To their horror and disbelief, web developers soon noticed that they couldn't safely host certain nominally harmless document types like *text/plain* or *text/csv* on behalf of their users; any attempt to do so would inevitably create a risk that such content could be misinterpreted as HTML.

Perhaps partly in response to these concerns, in 1999 the practice of unsolicited content sniffing was explicitly forbidden in HTTP/1.1:

> If and ***only*** if the media type is not given by a *Content-Type* field, the recipient may attempt to guess the media type via inspection of its content and/or the name extension(s) of the URI used to identify the resource.

Alas, this uncharacteristically clear requirement arrived a bit too late. Most browsers were already violating this rule to some extent, and absent a convenient way to gauge the potential consequences, their authors hesitated to simply ditch the offending code. Although several of the most egregious mistakes were cautiously reverted in the past decade, two companies—Microsoft and Apple—largely resisted the effort. They decided that interoperability with broken web applications should trump the obvious security problems. To pacify any detractors, they implemented a couple of imperfect, secondary security mechanisms intended to mitigate the risk.

Today, the patchwork of content-handling policies and the subsequently deployed restrictions cast a long shadow on the online world, making it nearly impossible to build certain types of web services without resorting to contrived and sometimes expensive tricks. To understand these limitations, let's begin by outlining several scenarios where a nominally passive document may be misidentified as HTML or something like it.

Document Type Detection Logic

The simplest and the least controversial type of document detection heuristics, and the one implemented by all modern browsers, is the logic implemented to handle the absence of the *Content-Type* header. This situation, which is encountered very rarely, may be caused by the developer accidentally omitting or mistyping the header name or the document being loaded over a non-HTTP transport mechanism such as *ftp:* or *file:*.

For HTTP specifically, the original RFCs explicitly permit the browser to examine the payload for clues when the *Content-Type* value is not available. For other protocols, the same approach is usually followed, often as a natural consequence of the design of the underlying code.

The heuristics employed to determine the type of a document typically amount to checking for static signatures associated with several dozen known file formats (such as images and common plug-in-handled files). The response will also be scanned for known substrings in order to detect signatureless formats such as HTML (in which case, the browser will look for familiar tags—*<body>*, *<font>*, etc). In many browsers, noncontent signals, such as trailing *.html* or *.swf* strings in the path segment of the URL, are taken into account as well.

The specifics of content-sniffing logic vary wildly from one browser to another and are not well documented or standardized. To illustrate, consider the handling of Adobe Flash (SWF) files served without *Content-Type*: In Opera, they are recognized unconditionally based on a content signature check; in Firefox and Safari, an explicit *.swf* suffix in the URL is required; and Internet Explorer and Chrome will not autorecognize SWF at all.

Rest assured, the SWF file format is not an exceptional case. For example, when dealing with HTML files, Chrome and Firefox will autodetect the document only if one of several predefined HTML tags appears at the very beginning of the file; while Firefox will be eager to "detect" HTML based solely on the presence of an *.html* extension in the URL, even if no recognizable markup is seen. Internet Explorer, on the other hand, will simply always default to HTML in the absence of *Content-Type*, and Opera will scan for known HTML tags within the first 1000 bytes of the returned payload.

The assumption behind all this madness is that the absence of *Content-Type* is an expression of an intentional wish by the publisher of the page—but that assumption is not always accurate and has caused a fair number of security bugs. That said, most web servers actively enforce the presence of a *Content-Type* header and will insert a default value if one is not explicitly generated by the server-side scripts that handle user requests. So perhaps there is no need to worry? Well, unfortunately, this is not where the story of content sniffing ends.

Malformed MIME Types

The HTTP RFC permits content sniffing only in the absence of *Content-Type* data; the browser is openly prohibited from second-guessing the intent of the webmaster if the header is present in any shape or form. In practice, however, this advice is not taken seriously. The next small step taken off the cliff was the decision to engage heuristics if the server-returned MIME type was deemed invalid in any way.

According to the RFC, the *Content-Type* header should consist of two slash-delimited alphanumeric tokens (*type/subtype*), potentially followed by other semicolon-delimited parameters. These tokens may contain any non-whitespace, seven-bit ASCII characters other than a couple of special "separators" (a generic set that includes characters such as "@", "?", and the slash itself). Most browsers attempt to enforce this syntax but do so inconsistently; the absence of a slash is seen almost universally as an invitation to content sniffing, and so is the inclusion of whitespaces and certain (but not all) control characters in the first portion of the identifier (the *type* token). On the other hand, the technically illegal use of high-bit characters or separators affects the validity of this field only in Opera.

The reasons for this design are difficult to understand, but to be fair, the security impact is still fairly limited. As far as web application developers are concerned, care must be exercised not to make typos in *Content-Type* values and not to allow users to specify arbitrary, user-controlled MIME types (merely validated against a blacklist of known bad options). These requirements may be unexpected, but usually they do not matter a lot. So, what are we ultimately getting at?

Special Content-Type Values

The first clear signal that content sniffing was becoming truly dangerous was the handling of a seemingly unremarkable MIME type known as *application/octet-stream*. This specific value is not mentioned at all in the HTTP specification but is given a special (if vague) role deep in the bowels of RFC 2046:[1]

> The recommended action for an implementation that receives an *application/octet-stream* entity is to simply offer to put the data in a file, with any *Content-Transfer-Encoding* undone, or perhaps to use it as input to a user-specified process.

The original intent of this MIME type may not be crystal clear from the quoted passage alone, but it is commonly interpreted as a way for web servers to indicate that the returned file has no special meaning to the server and that it should not have one to the client. Consequently, most web servers default to *application/octet-stream* on all types of opaque, nonweb files, such as downloadable executables or archives, if no better *Content-Type* match can be found. However, in rare cases of administrator errors (for example, due to deletion of the essential *AddType* directives in Apache configuration files), web servers may also fall back to this MIME type on documents meant for in-browser consumption. This configuration error is, of course, very easy to detect and fix, but Microsoft, Opera, and Apple nevertheless chose to compensate for it. The browsers from these vendors eagerly engage in content sniffing whenever *application/octet-stream* is seen.*

This particular design decision has suddenly made it more difficult for web applications to host binary files on behalf of the user. For example, any code-hosting platform must exercise caution when returning executables or source archives as *application/octet-stream*, because there is a risk they may be misinterpreted as HTML and displayed inline. That's a major issue for any software hosting or webmail system and for many other types of web apps. (It's slightly safer for them to use any other generic-sounding MIME type, such as *application/binary*, because there is no special case for it in the browser code.)

In addition to the special treatment given to *application/octet-stream*, a second, far more damaging exception exists for *text/plain*. This decision, unique to Internet Explorer and Safari, traces back to RFC 2046. In that document, *text/plain* is given a dual function: first, to transmit plaintext documents (ones that *"do not provide for or allow formatting commands, font attribute specifications, processing instructions, interpretation directives, or content markup"*) and, second, to provide a fallback value for any text-based documents not otherwise recognized by the sender.

* In Internet Explorer, this implemented logic differs subtly from a scenario where no *Content-Type* is present. Instead of always assuming HTML, the browser will scan the first 256 bytes for popular HTML tags and other predefined content signatures. From the security standpoint, however, it's not a very significant difference.

The distinction between *application/octet-stream* and *text/plain* fallback made perfect sense for email messages, a topic that this RFC originally dealt with, but proved to be much less relevant to the Web. Nevertheless, some web servers adopted *text/plain* as the fallback value for certain types of responses (most notably, the output of CGI scripts).

The *text/plain* logic subsequently implemented in Internet Explorer and Safari in order to detect HTML in such a case is really bad news: It robs web developers of the ability to safely use this MIME type to generate user-specific plaintext documents and offers no alternatives. This has resulted in a substantial number of web application vulnerabilities, but to this day, Internet Explorer developers seem to have no regrets and have not changed the default behavior of their code.

Safari developers, on the other hand, recognized and tried to mitigate the risk while keeping the functionality in place—but they failed to appreciate the complexity of the Web. The solution implemented in their browser is to rely on a secondary signal in addition to the presence of a plausible-looking HTML markup in the document body. The presence of an extension such as *.html* or *.xml* at the end of the URL path is interpreted by their implementation as a sign that content sniffing can be performed safely. After all, the owner of the site wouldn't name the file this way otherwise, right?

Alas, the signal they embraced is next to worthless. As it turns out, almost all web frameworks support at least one of several methods for encoding parameters in the path segment of the URL instead of in the more traditionally used query part. For example, in Apache, one such mechanism is known as PATH_INFO, and it happens to be enabled by default. By leveraging such a parameter-passing scheme, the attacker can usually append nonfunctional garbage to the path, thereby confusing the browser without affecting how the server will respond to the submitted request itself.

To illustrate, the following two URLs will likely have the same effect for websites running on Apache or IIS:

```
http://www.fuzzybunnies.com/get_file.php?id=1234
```

and

```
http://www.fuzzybunnies.com/get_file.php/evil.html?id=1234
```

In some less-common web frameworks, the following approach may also work:

```
http://www.fuzzybunnies.com/get_file.php;evil.html?id=1234
```

Unrecognized Content Type

Despite the evident trouble with *text/plain*, the engineers working on Internet Explorer decided to take their browser's heuristics even further. Internet Explorer applies both content sniffing and extension matching* not only to a handful of generic MIME types but also to any document type not immediately recognized by the browser. This broad category may include everything from JSON (*application/json*) to multimedia formats such as Ogg Vorbis (*audio/ogg*).

Such a design is, naturally, problematic and causes serious problems when hosting any user-controlled document formats other than a small list of universally supported MIME types registered internally in the browser or when routed to a handful of commonly installed external applications.

Nor do the content-sniffing habits of Internet Explorer finally end there: The browser will also resort to payload inspection when dealing with internally recognized document formats that, for any reason, can't be parsed cleanly. In Internet Explorer versions prior to 8, serving a user-supplied but non-validated file claiming to be an JPEG image can lead to the response being treated as HTML. And it gets even more hilarious: Even a subtle mistake, such as serving a valid GIF file with *Content-Type: image/jpeg*, triggers the same code path. Heck, several years ago, Internet Explorer even detected HTML on any valid, properly served PNG file. Thankfully, this logic has since been disabled—but the remaining quirks are still a minefield.

NOTE *In order to fully appreciate the risk of content sniffing on valid images, note that it is not particularly difficult to construct images that validate correctly but that carry attacker-selected ASCII strings—such as HTML markup—in the raw image data. In fact, it is relatively easy to construct images that, when scrubbed, rescaled, and recompressed using a known, deterministic algorithm, will have a nearly arbitrary string appear out of the blue in the resulting binary stream.*

To its credit, in Internet Explorer 8 and beyond, Microsoft decided to disallow most types of gratuitous content sniffing on known MIME types in the *image/** category. It also disallowed HTML detection (but not XML detection) on image formats not recognized by the browser, such as *image/jp2* (JPEG2000).

This single tweak aside, Microsoft has proven rather unwilling to make meaningful changes to its content-sniffing logic, and its engineers have publicly defended the need to maintain compatibility with broken websites.[2] Microsoft probably wants to avoid the wrath of large institutional customers, many of whom rely on ancient and poorly designed intranet apps and depend on the quirks of the Internet Explorer–based monoculture on the client end.

In any case, due to the backlash that Internet Explorer faced over its *text/plain* handling logic, newer versions offer a partial workaround: an optional

* Naturally, path-based extension matching is essentially worthless for the reasons discussed in the previous section; but in the case of Internet Explorer 6, it gets even worse. In this browser, the extension can appear in the query portion of the URL. Nothing stops the attacker from simply appending *?foo=bar.html* to the requested URL, effectively ensuring that this check is always satisfied.

HTTP header, *X-Content-Type-Options: nosniff*, which allows website owners to opt out of most of the controversial content heuristics. The use of this header is highly recommended; unfortunately, the support for it has not been backported to versions 6 and 7 of the browser and has only a limited support in other browsers. In other words, it cannot be depended on as a sole defense against content sniffing.

NOTE *Food for thought: According to the data collected in a 2011 survey by SHODAN and Chris John Riley,[3] only about 0.6 percent of the 10,000 most popular websites on the Internet used this header on a site-wide level.*

Defensive Uses of Content-Disposition

The *Content-Disposition* header, mentioned several times in Part I of this book, may be considered a defense against content sniffing in some use cases. The function of this header is not explained satisfactorily in the HTTP/1.1 specification. Instead, it is documented only in RFC 2183,[4] where its role is explained only as it relates to mail applications:

> Bodyparts can be designated "attachment" to indicate that they are separate from the main body of the mail message, and that their display should not be automatic, but contingent upon some further action of the user. The MUA* might instead present the user of a bitmap terminal with an iconic representation of the attachments, or, on character terminals, with a list of attachments from which the user could select for viewing or storage.

The HTTP RFC acknowledges the use of *Content-Disposition: attachment* in the web domain but does not elaborate on its intended function. In practice, upon seeing this header during a normal document load, most browsers will display a file download dialog, usually with three buttons: "open," "save," and "cancel." The browser will not attempt to interpret the document any further unless the "open" option is selected or the document is saved to disk and then opened manually. For the "save" option, an optional *filename* parameter included in the header is used to suggest the name of the download, too. If this field is absent, the filename will be derived from the notoriously unreliable URL path data.

Because the header prevents most browsers from immediately interpreting and displaying the returned payload, it is particularly well suited for safely hosting opaque, downloadable files such as the aforementioned case of archives or executables. Furthermore, because it is ignored on type-specific subresource loads (such as *<img>* or *<script>*), it may also be employed to protect user-controlled JSON responses, images, and so on against content sniffing risks. (The reason why all implementations ignore *Content-Disposition* for these types of navigation is not particularly clear, but given the benefits, it's best not to question the logic now.)

* MUA stands for "mail user agent," that is, a client application used to retrieve, display, and compose mail messages.

One example of a reasonably robust use of *Content-Disposition* and other HTTP headers to discourage content sniffing on a JSON response may be

```
Content-Type: application/json; charset=utf-8
X-Content-Type-Options: nosniff
Content-Disposition: attachment; filename="json_response.txt"

{ "search_term": "<html><script>alert('Hi mom!')</script>", ... }
```

The defensive use of *Content-Disposition* is highly recommended where possible, but it is important to recognize that the mechanism is neither mandated for all user agents nor well documented. In less popular browsers, such as Safari Mobile, the header may have no effect; in mainstream browsers, such as Internet Explorer 6, Opera, and Safari, a series of *Content-Disposition* bugs have at one point or another rendered the header ineffective in attacker-controlled cases.

Another problem with the reliance on *Content-Disposition* is that the user may still be inclined to click "open." Casual users can't be expected to be wary of viewing Flash applets or HTML documents just because a download prompt gets in the way. In most browsers, selecting "open" puts the document in a *file:* origin, which may be problematic on its own (the recent improvements in Chrome certainly help), and in Opera, the document will be displayed in the context of the originating domain. Arguably, Internet Explorer makes the best choice: HTML documents are placed in a special sandbox using a *mark-of-the-web* mechanism (outlined in more detail in Chapter 15), but even in that browser, Java or Flash applets will not benefit from this feature.

Content Directives on Subresources

Most content-related HTTP headers, such as *Content-Type*, *Content-Disposition*, and *X-Content-Type-Options*, have largely no effect on type-specific subresource loads, such as <*img*>, <*script*>, or <*embed*>. In these cases, the embedding party has nearly complete control over how the response will be interpreted by the browser.

Content-Type and *Content-Disposition* may also not be given much attention when handling requests initiated from within plug-in-executed code. For example, recall from Chapter 9 that any *text/plain* or *text/csv* documents may be interpreted by Adobe Flash as security-sensitive *crossdomain.xml* policies unless an appropriate site-wide metapolicy is present in the root directory on the destination server. Whether you wish to call it "content sniffing" or just "content-type blindness," the problem is still very real.

Consequently, even when all previously discussed HTTP headers are used religiously, it is important to always consider the possibility that a third-party page may trick the browser into interpreting that page as one of several problematic document types; applets and applet-related content, PDFs, style-sheets, and scripts are usually of particular concern. To minimize the risk of mishaps, you should carefully constrain the structure and character set of any served payloads or use "sandbox" domains to isolate any document types that can't be constrained particularly well.

Downloaded Files and Other Non-HTTP Content

The behavior of HTTP headers such as *Content-Type*, *Content-Disposition*, and *X-Content-Type-Options* may be convoluted and exception ridden, but at the very least, they add up to a reasonably consistent whole. Still, it is easy to forget that in many real-world cases, the metadata contained in these headers is simply not available—and in that case, all bets are off. For example, the handling of documents retrieved over *ftp:*, or saved to disk and opened over the *file:* protocol, is highly browser- and protocol-specific and often surprises even the most seasoned security experts.

When opening local files, browsers usually give precedence to file extension data, and if the extension is one of the hardcoded values known to the browser, such as *.txt* or *.html*, most browsers will take this information at face value. Chrome is the exception; it will attempt to autodetect certain "passive" document types, such as JPEG, even inside *.txt* documents. (HTML, however, is strictly off-limits.)

When it comes to other extensions registered to external programs, the behavior is a bit less predictable. Internet Explorer will usually invoke the external application, but most other browsers will resort to content sniffing, behaving as though they loaded the document over HTTP with no *Content-Type* set. All browsers will also fall back to content sniffing if the extension is not known (say, *.foo*).

The heavy reliance on file extension data and content sniffing for *file:* documents creates an interesting contrast with the normal handling of Internet-originating resources. On the Web, *Content-Type* is by and large the authoritative descriptor of document type. File extension information is ignored most of the time, and it is perfectly legal to host a functional JPEG file at a location such as *http://fuzzybunnies.com/gotcha.txt*. But what happens when this document is downloaded to disk? Well, in such case, the effective meaning of the resource will unexpectedly change: When accessing it over the *file:* protocol, the browser may insist on rendering it as a text file, based strictly on the extension data.

The example above is fairly harmless, but other content promotion vectors, such as an image becoming an executable, may be more troubling. To that effect, Opera and Internet Explorer will attempt to modify the extension to match the MIME type for a handful of known *Content-Type* values. Other browsers do not offer this degree of protection, however, and may even be thoroughly confused by the situation they find themselves in. Figure 13-1 captures Firefox in one such embarrassing moment.

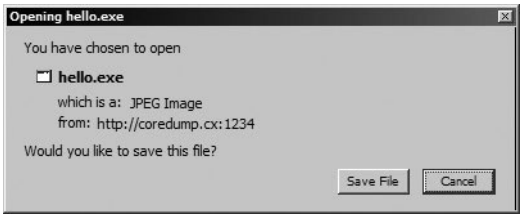


Figure 13-1: Prompt displayed by Firefox when saving a Content-Type: image/jpeg *document served with* Content-Disposition: attachment. *The "hello.exe" filename is derived by the browser from a nonfunctional PATH_INFO suffix appended by the attacker at the end of the URL. The prompt incorrectly claims that the* .exe *file is a "JPEG Image." In fact, when saved to disk, it will be an executable.*

This problem underscores the importance of returning an explicit, harmless *filename* value whenever using a *Content-Disposition* attachment, to prevent the victim from being tricked into downloading a document format that the site owner never intended to host.

Given the complex logic used for *file:* URLs, the simplicity of *ftp:* handling may come as a shock. When accessing documents over FTP, most browsers pay no special attention to file extensions and will simply indulge in rampant content sniffing. One exception is Opera, where extension data still takes precedence. From the engineering point of view, the prevalent approach to FTP may seem logical: The protocol can be considered roughly equivalent to HTTP/0.9. Nevertheless, the design also violates the principle of least astonishment. Server owners would not expect that by allowing users to upload *.txt* documents to an FTP site, they are automatically consenting to host active HTML content within their domain.

Character Set Handling

Document type detection is one of the more important pieces of the content-processing puzzle, but it is certainly not the only one. For all types of text-based files rendered in the browser, one more determination needs to be made: The appropriate character set transformation must be identified and applied to the input stream. The output encoding sought by the browser is typically UTF-8 or UTF-16; the input, on the other hand, is up to the author of the page.

In the simplest scenario, the appropriate encoding method will be provided by the server in a *charset* parameter of the *Content-Type* header. In the case of HTML documents, the same information may also be conveyed to some extent through the <*meta*> directive. (The browser will attempt to speculatively extract and interpret this directive before actually parsing the document.)

Unfortunately, the dangerous qualities of certain character encodings, as well as the actions taken by the browser when the *charset* parameter is not present or is not recognized, once again make life a lot more interesting than the aforementioned simple rule would imply. To understand what can go wrong, we first need to recognize three special classes of character sets that may alter the semantics of HTML or XML documents:

- **Character sets that permit noncanonical representations of standard 7-bit ASCII codes.** Such noncanonical sequences could be used to cleverly encode HTML syntax elements, such as angle brackets or quotes, in a manner that survives a simple server-side check. For example, the famously problematic UTF-7 encoding permits the "<" character to be encoded as a five-character sequence of "+ADw-", a string that most server-side filters will happily permit as is. In a similar vein, UTF-8 specification formally prohibits, but technically permits, "<" to be represented by unnecessarily verbose 2- to 5-byte sequences, from 0xC0 0xBC to 0xFC 0x80 0x80 0x80 0x80 0xBC.*

* Today, this problem is mitigated by most browsers: Their parsers now have additional checks to reject overlong UTF-8 encodings as a matter of principle. The same cannot be said of all possible server-side UTF-8 libraries, however.

- **Variable length encodings that give special meaning to one or more bytes that follow a special prefix.** Such logic may result in legitimate HTML syntax elements being "consumed" as part of an unintentional multibyte literal. For example, the Shift JIS prefix code 0xE0 can cause the subsequent angle bracket or a quote to be consumed in Internet Explorer, Firefox, and Opera (but not in Chrome), possibly severely altering the meaning of the inline markup.

  The opposite problem may also occur: The server may be convinced that it is outputting a multibyte literal, but this literal may be rejected by the browser and interpreted as several individual characters. In EUC-KR, the 0x8E prefix is honored only if the subsequent character has an ASCII code of 0x41 or higher. Any less and it will not have the expected effect, but not all server-side implementations may notice.
- **Encodings that are completely incompatible with 8-bit ASCII.** These cases will simply lead to a very different view of document structure between the client and the server. Common examples include UTF-16 or UTF-32.

The bottom line is that unless the server has a perfect command of the character set it is generating and unless it is certain that the client will not apply an unexpected transformation to the payload, serious complications may arise. For example, consider a web application that removes angle brackets from the highlighted user-controlled string in the following piece of HTML:

```
You are currently viewing:
<span class="blog_title">
 +ADw-script+AD4-alert("Hi mom!")+ADw-/script+AD4-
</span>
```

If that document is interpreted as UTF-7 by the receiving party, the actual parsed markup will look as follows:

```
You are currently viewing:
<span class="blog_title">
 <script>alert("Hi mom!")</script>
</span>
```

A similar problem, this time related to byte consumption in Shift JIS encoding, is illustrated below. A multibyte prefix is permitted to consume a closing quote, and as a result, the associated HTML tag is not terminated as expected, enabling the attacker to inject an extra *onerror* handler into the markup:

```
<img src="http://fuzzybunnies.com/[0xE0]">
 ...this is still a part of the markup...
  ...but the server doesn't know...
  " onerror="alert('This will execute!')"
<div>
  ...page content continues...
</div>
```

It is simply imperative to prevent character set autodetection for all text-based documents that contain any type of user-controlled data. Most browsers will engage in character set detection if the *charset* parameter is not found in the *Content-Type* header or in the *<meta>* tag. Some marked differences exist between the implementations (for example, only Internet Explorer is keen to detect UTF-7), but you should never assume that the outcome of character set sniffing will be safe.

Character set autodetection will also be attempted if the character set is not recognized or is mistyped; this problem is compounded by the fact that charset naming can be ambiguous and that web browsers are inconsistent in how much tolerance they have for common name variations. As a single data point, consider the fact that Internet Explorer recognizes both ISO-8859-2 and ISO8859-2 (with no dash after the ISO part) as valid character set identifiers in the *Content-Type* header but fails to recognize UTF8 as an alias for UTF-8. The wrong choice can cause some serious pain.

NOTE *Fun fact: The* X-Content-Type-Options *header has no effect on character-sniffing logic.*

Byte Order Marks

We are not done with character set detection just yet! Internet Explorer needs to be singled out for yet another dramatically misguided content-handling practice: the tendency to give precedence to the so-called *byte order mark (BOM)*, a sequence of bytes that can be placed at the beginning of a file to identify its encoding, over the explicitly provided *charset* data. When such a marker is detected in the input file, the declared character set is ignored.

Table 13-1 shows several common markers. Of these, the printable UTF-7 BOM is particularly sneaky.

Table 13-1: Common Byte Order Markers (BOMs)

Encoding name	Byte order mark sequence
UTF-7	"+/v" followed by "8", "9", "+", or "/"
UTF-8	0xEF 0xBB 0xBF
UTF-16 little endian	0xFF 0xFE
UTF-16 big endian	0xFE 0xFF
UTF-32 little endian	0xFF 0xFE 0x00 0x00
UTF-32 big endian	0x00 0x00 0xFE 0xFF
GB-18030	0x84 0x31 0x95 0x33

NOTE *Microsoft engineers acknowledge the problem with this design and, as of this writing, say that the logic may be revised, depending on the outcome of compatibility tests. If the problem is resolved by the time this book hits the shelves, kudos to them. Until then, allowing the attacker to control the first few bytes of an HTTP response that is not otherwise protected by* Content-Disposition *may be a bad idea—and other than padding the response, there is no way to work around this glitch.*

Character Set Inheritance and Override

Two additional, little-known mechanisms should be taken into account when evaluating the potential impact on character set handling strategies in contemporary web browsers. Both of these features may permit an attacker to force undesirable character encoding upon another page, without relying on character sniffing.

The first apparatus in question, supported by all but Internet Explorer, is known as *character set inheritance*. Under this policy, any encoding defined for the top-level frame may be automatically applied to any framed documents that do not have their own, valid *charset* value set. Initially, such inheritance is extended to all framing scenarios, even across completely unrelated websites. However, when Stefan Esser, Abhishek Arya, and several other researchers demonstrated a number of plausible attacks that leveraged this feature to force UTF-7 parsing on unsuspecting targets, Firefox and WebKit developers decided to limit the behavior to same-origin frames. (Opera still permits cross-domain inheritance. Although it does not support UTF-7, other problematic encodings, such as Shift JIS, are fair game.)

The other mechanism that deserves mention is the ability to manually override the currently used character set. This feature is available through the *View* > *Encoding* menu or similar in most browsers. Using this menu to change the character set causes the page and all its subframes (including cross-domain ones!) to be reparsed using the selected encoding, regardless of any *charset* directives encountered earlier for that content.

Because users may be easily duped into selecting an alternative encoding for an attacker-controlled page (simply in order to view it correctly), this design should make you somewhat uncomfortable. Casual users can't be expected to realize that their election will also apply to hidden <*iframe*> tags and that such a seemingly innocuous action may enable cross-site scripting attacks against unrelated web properties. In fact, let's be real: Most of them will not know—and should not have to know—what an <*iframe*> is.

Markup-Controlled Charset on Subresources

We are nearing the end of the epic journey through the web of content-handling quirks, but we are not quite done yet. Astute readers may recall that in "Type-Specific Content Inclusion" on page 82, I mentioned that on certain types of subresources (namely, stylesheets and scripts), the embedding page can specify its own *charset* value in order to apply a specific transformation to the retrieved document, for example,

```
<script src="http://fuzzybunnies.com/get_js_data.php" charset="EUC-JP">
```

This parameter is honored by all browsers except for Opera. Where it is supported, it typically does not take precedence over *charset* in *Content-Type*, unless that second parameter is missing or unrecognized. But to every rule, there is an exception, and all too often, the name of this exception is Internet Explorer 6. In that still-popular browser, the encoding specified by the markup overrides HTTP data.

Does this behavior matter in practice? To fully grasp the consequences, let's also quickly return to Chapter 6, where we debated the topic of securing server-generated, user-specific, JSON-like code against cross-domain inclusion. One example of an application that needs such a defense is a searchable address book in a webmail application: The search term is provided in the URL, and a JavaScript serialization of the matching contacts is returned to the browser but must be shielded from inclusion on unrelated sites.

Now, let's assume that the developer came up with a simple trick to prevent third-party web pages from loading this data through *<script src=…>*: A single "*//*" prefix is used to turn the entire response into a comment. Same-origin callers that use the *XMLHttpRequest* API can simply examine the response, strip the prefix, and pass the data to *eval(…)*—but remote callers, trying to abuse the *<script src=…>* syntax, will be out of luck.

In this design, a request to */contact_search.php?q=smith* may yield the following response:

```
// var result  = { "q": "smith", "r": [ "j.smith@example.com" ] };
```

As long as the search term is properly escaped or filtered, this scheme appears safe. But when we realize that the attacker may force the response to be interpreted as UTF-7, the picture changes dramatically. A seemingly benign search term that, as far as the server is concerned, contains no illegal characters could still unexpectedly decode to

```
// var result = { "q": "smith[CR][LF]
var gotcha = { "", "r": [ "j.smith@example.com" ] };
```

This response, when loaded via *<script src=… charset=utf-7>* inside the victim's browser, gives the attacker access to a portion of the user's address book.

This is not just a thought exercise: The "*//*" approach is fairly common on the Web, and Masato Kinugawa, a noted researcher, found several popular web applications affected by this bug. And a more contrived variant of the same attack is also possible against other execution-preventing prefixes, such as *while (1);*. In the end, the problems with cross-domain *charset* override on *<script>* tags is one of the reasons why in Chapter 6, we strongly recommend using a robust parser-stopping prefix to prevent the interpreter from ever looking at any attacker-controlled bits. Oh—and if you factor in the support for E4X, the picture becomes even more interesting,[5] but let's leave it at that.

Detection for Non-HTTP Files

To wrap up this chapter, let's look at the last missing detail: character set encoding detection for documents delivered over non-HTTP protocols. As can be expected, documents saved to disk and subsequently opened over the *file:* protocol, or loaded by other means where the usual *Content-Type* metadata is absent, will usually be subjected to character set detection logic.

However, unlike with document determination heuristics, there is no substantial difference among all the possible delivery methods: In all cases, the sniffing behavior is roughly the same.

There is no clean and portable way to address this problem for all text-based documents, but for HTML specifically, the impact of character set sniffing can be mitigated by embedding a *<meta>* directive inside the document body:

```
<meta http-equiv="Content-Type" content="text/html;charset=...">
```

You should not ditch *Content-Type* in favor of this indicator. Unlike *<meta>*, the header works for non-HTML content, and it is easier to enforce and audit on a site-wide level. That said, documents that are likely to be saved to disk and that contain attacker-controlled tidbits will benefit from a redundant *<meta>* tag. (Just make sure that this value actually matches *Content-Type*.)

Security Engineering Cheat Sheet

Good Security Practices for All Websites

- ☑ Instruct the web server to append the *X-Content-Options: nosniff* header to all HTTP responses.
- ☑ Consult the cheat sheet in Chapter 9 to set up an appropriate */crossdomain.xml* meta-policy.
- ☑ Configure the server to append default *charset* and *Content-Type* values on all responses that would otherwise not have one.
- ☑ If you are not using path-based parameter passing (such as PATH_INFO), consider disabling this feature.

When Generating Documents with Partly Attacker-Controlled Contents

- ☑ Always return an explicit, valid, well-known *Content-Type* value. Do not use *text/plain* or *application/octet-stream*.
- ☑ For any text-based documents, return a explicit, valid, well-known *charset* value in the *Content-Type* header; UTF-8 is preferable to any other variable-width encodings. Do not assume that *application/xml+svg*, *text/csv*, and other non-HTML documents do not need a specified character set. For HTML, consider a redundant *<meta>* directive if it's conceivable that the file may be downloaded by the user. Beware of typos—UTF8 is not a valid alias for UTF-8.
- ☑ Use *Content-Disposition: attachment* and an appropriate, explicit *filename* value for responses that do not need to be viewed directly—including JSON data.
- ☑ Do not allow the user to control the first few bytes of the file. Constrain the response as much as possible. Do not pass through NULs, control characters, or high-bit values unless absolutely necessary.
- ☑ When performing server-side encoding conversions, be sure that your converters reject all unexpected or invalid inputs (e.g., overlong UTF-8).

When Hosting User-Generated Files

Consider using a sandbox domain if possible. If you intend to host unconstrained or unknown file formats, a sandbox domain is a necessity. Otherwise, at the very minimum, do the following:

- ☑ Use *Content-Disposition: attachment* and an appropriate, explicit *filename* value that matches the *Content-Type* parameter.
- ☑ Carefully validate the input data and always use the appropriate, commonly recognized MIME type. Serving JPEG as *image/gif* may lead to trouble. Refrain from hosting MIME types that are unlikely to be supported by popular browsers.
- ☑ Refrain from using *Content-Type: application/octet-stream* and use *application/binary* instead, especially for unknown document types. Refrain from returning *Content-Type: text/plain*. Do not permit user-specified *Content-Type* headers.

14

DEALING WITH ROGUE SCRIPTS

In the previous five chapters, we examined a fairly broad range of browser security mechanisms—and looking back at them, it is fair to say that almost all share a common goal: to stop rogue content from improperly interfering with any other, legitimate web pages displayed in a browser. This is an important pursuit but also a fairly narrow one; subverting the boundaries between unrelated websites is a large part of every attacker's repertoire but certainly not the only trick in the book.

The other significant design-level security challenge that all browsers have to face is that attackers may abuse well-intentioned scripting capabilities in order to disrupt or impersonate third-party sites without actually interacting with the targeted content. For example, if JavaScript code controlled by an attacker is permitted to create arbitrary undecorated windows on a screen, the attacker may find that, rather than look for a way to inject a malicious payload into the content served at *fuzzybunnies.com*, it may be easier to just open a window with a believable replica of the address bar, thus convincing the user that the content displayed is from a trusted site.

Unfortunately for victims, in the early days of the Web, no real attention was given to the susceptibility of JavaScript APIs to attacks meant to disrupt or confuse users, and, unlike cross-domain content isolation issues, this class of problems is still not taken very seriously. The situation is unlikely to change anytime soon: Vendor resources are stretched thin between addressing comparatively more serious implementation-level flaws in the notoriously buggy browser codebases and rolling out new, shiny security features that appease web application developers, users, and the mainstream press alike.

Denial-of-Service Attacks

The possibility of an attacker crashing a browser or otherwise rendering it inoperable is one of the most common, obvious, and least appreciated issues affecting the modern Web. In the era of gadgets and mashups, it can have unexpectedly unpleasant consequences, too.

The most prominent reason why most browsers are susceptible to *denial-of-service (DoS)* attacks is due simply to a lack of planning: Neither the underlying document formats nor the capabilities exposed through scripting languages were designed to have a sensible, constrained worst-case CPU or memory footprint. In other words, any sufficiently complex HTML file or an endless JavaScript loop could bring the underlying operating system to its knees. Worse, the attempts to mandate resource limits or to give users a way to resume control of a runaway browser following a visit to a rogue page meet with resistance. For example, the authors of many of the recently proposed HTML5 APIs provide no advice on preventing resource exhaustion attacks, nor do they even acknowledge this need, because they think that any limits imposed today will likely hinder the growth of the Web 5 or 10 years from now. Browser developers, in turn, refuse to take any action absent any standards-level guidance.

A common utilitarian argument against any proposed DoS defenses is that they are pointless—that the browser is hopelessly easy to crash in a multitude of ways, so why take special measures to address a specific vector today? It's hard to argue with this view, but it's also important to note that it acts as a self-fulfilling prophecy: The steady increase in the number of DoS vectors is making it more and more unlikely that the situation will be comprehensively addressed any time soon.

NOTE *To be fair, the computational complexity of certain operations is not the only reason why browsers are easy to crash. Vendors are also constrained by the need to maintain a significant degree of synchronicity during page-rendering and script-execution steps (see Chapter 6). This design eliminates the need for website developers to write reentrant and thread-safe code and has substantial code complexity and security benefits. Unfortunately, it also makes it much easier for one document to lock up the entire browser, or at least a good portion thereof.*

Regardless of all these considerations, and even if browser vendors refuse to acknowledge DoS risks as a specific flaw, the impact of such attacks is difficult to ignore. For one, whenever a browser is brought down, there is a

substantial risk of data loss (in the browser itself or in any applications indirectly affected by the attack). Also, on some social-networking sites, an attacker may be able to lock out the victim from the site simply by sharing a rogue gadget, or perhaps even a well-selected image, with the victim, preventing that person from ever using that service again.

Some of the common tricks used to take a browser out of service include loading complex XHTML or SVG documents, opening a very large number of windows, running an endless JavaScript loop that allocates memory, queuing a significant number of *postMessage(...)* calls, and so on. While these examples are implementation-specific, every browser offers a fair number of ways to achieve this goal. Even in Chrome, which uses separate renderer processes to isolate unrelated pages, it's not difficult to bring down the entire browser: The top-level process mediates a variety of script-accessible and sometimes memory- or CPU-intensive tasks.

Given the above, it's no surprise that despite generally dismissive attitudes, the major browsers nevertheless implement several DoS countermeasures. They do not add up to a coherent strategy, and have they have been rolled out only in response to the widespread abuse of specific APIs or to mitigate nonmalicious but common programming errors. Nevertheless, let's look at them briefly.

Execution Time and Memory Use Restrictions

Because of the aforementioned need to enforce a degree of synchronicity for many types of JavaScript operations, most browser vendors err on the side of caution and execute scripts synchronously with most of the remaining browser code. This design has an obvious downside: A good portion of the browser may become completely unresponsive as the JavaScript engine is, say, trying to evaluate a bogus *while (1)* loop. In Opera and Chrome, the top-level user interface will still be largely responsive, if sluggish, but in most other browsers, it won't even be possible to close the browser window using the normal UI.

Because endless loops are fairly easy to create by accident, in order to aid developers, Internet Explorer, Firefox, Chrome, and Safari enforce a modest time limit on any continuously or nearly continuously executing scripts. If the script is making the browser unresponsive for longer than a couple of seconds, the user will be shown a dialog and given the option to abort execution. Picking this option will have a result similar to encountering an unhandled exception, that is, of abandoning the current execution flow.

Regrettably, such a limit is not a particularly robust defense against malicious scripts. For example, regardless of the user's choice, it is still possible to resume execution through timers or event handlers, and it's easy to avoid triggering the prompt in the first place by periodically returning the CPU briefly to an idle state in order to reset the counter. Too, as noted previously, there are ways to hog CPU resources without resorting to busy loops: Rendering complex XHTML, SVG, or XSLT documents can be just as disruptive and is not subject to any checks.

Execution time aside, there have been attempts to control the memory footprint of executed scripts. The size of the call stack is limited to a browser-specific value between 500 and 65535, and attempting a deeper recursion will result in an unconditional stop. Script heap size, on the other hand, is typically not restricted in a meaningful way; pages can allocate and use up gigabytes of memory. In fact, most of the previously implemented restrictions (such as the 16MB cap in Internet Explorer 6) have been removed in more recent releases.

Connection Limits

In many web applications, each web page consists not only of the proper HTML document retrieved from the URL visible in the address bar but also as many as several dozen other, separately loaded subresources, such as images, stylesheets, and scripts. Because requesting all of these elements through individually established HTTP connections can be slow, the reader may recall from Chapter 3 that the protocol has been extended to offer keep-alive sessions and request pipelining. But even with these improvements, one stubborn problem remains. The inherent limitation of the protocol is that the server must always send responses in the same order that it received the requests, so if any of the subresources (no matter how inconsequential) takes a bit longer to generate, the loading of all subsequent ones will be delayed.

To work around this problem, and to optimize performance when keep-alive requests or pipelining can't be used, all browsers permit the opening of several simultaneous HTTP connections to the destination server. This way, the browser can issue multiple requests in parallel.

Unfortunately, the parallel connection design can be expensive for the destination website, especially if the server relies on the traditional *fork()*-based connection-handling architecture.[*] Therefore, in order to limit the risk of accidentally or intentionally launching a distributed DoS attack, the number of parallel connections needs to be limited to a modest per-host value, typically between 4 and 16. Furthermore, to prevent attackers from overloading the browser itself (or affecting the performance of the nearby networking equipment), the total number of simultaneous connections to all destinations is also constrained to a low multiple of the per-host cap.

NOTE *In many implementations, the per-host connection limit is enforced by looking at DNS labels, not at IP addresses. Therefore, an attacker may still be able to point several bogus DNS entries in his own domains to any unrelated target IP and circumvent the first restriction. The global connection limit will be still in effect, though.*

Although the number of concurrent HTTP sessions is limited, there are no practical restrictions on how long an active session may be kept alive (that is, as long as no kernel-level TCP/IP timeouts are encountered). This design

[*] The traditional design of most Unix services is to have a master "listener" process, and then create a new process for handling every accepted connection. For the developer, this model is remarkable in its simplicity; but it comes with many significant hidden costs for the operating system, which sometimes finds handling more than several hundred simultaneous connections at once challenging.

may make it possible for attackers to simply exhaust the global connection limit by talking to a couple of intentionally slow destinations, preventing the user from doing anything useful in the meantime.

Pop-Up Filtering

The *window.open(...)* and *window.showModalDialog(...)*[*] APIs permit web pages to create new browser windows, pointing them to any otherwise permitted URLs. In both cases, the browser may be instructed not to show certain window decorations for the newly loaded document or to position the window on the screen in a specific way. A simple use of *window.open(...)* might look like this:

```
window.open("/hello.html", "_blank", "menubar=no,left=50,top=50");
```

In addition to these two JavaScript methods, new windows may also be opened indirectly by programatically interacting with certain HTML elements. For example, it is possible to call the *click()* method on an HTML link or to invoke the *submit()* method on a form. If the relevant markup includes a *target* parameter, the resulting navigation will take place in a new window of a specified name.

As could be expected, the ability for random web pages to open new browser windows soon proved to be problematic. In the late 1990s, many players in the then-young online advertising industry decided they needed to attract attention to their ads at any cost, even at the expense of profoundly annoying and alienating their audiences. Automatically spawning windows solely to show a flashy advertisement seemed like a great way to do business and make new friends.

Pop-up and pop-under[†] advertisements have quickly emerged as one of the best-known and most reviled aspects of the Web. For good reason, too: Especially with pop-unders, it would not be unusual to amass a dozen of them after two to three hours of casual browsing.

Due to widespread complaints, browser vendors stepped in and implemented a simple restriction: Spurious attempts by non-whitelisted pages to create new windows would be silently ignored.[‡] Exceptions were made for attempts made immediately after a mouse click or a similar user action. For

[*] The little-known *showModalDialog(...)* method is a bit of a misnomer. It is essentially equivalent to *window.open(...)*, but it is supposed to vaguely emulate the behavior of a modal dialog by blocking the scripts in the calling context until such a "dialog" window is dismissed. The exact behavior of this API varies randomly from one browser to another. For example, it is sometimes possible for other pages to navigate the underlying window or execute new scripts while the original JS code that called *showModalDialog(...)* is in progress.

[†] A "pop-under" is a pop-up window that, immediately after its creation, is moved to the back of the window stack with the help of *opener.window.focus()* or *window.blur()*. Pop-unders are arguably slightly less distracting than pop-ups, because the user does not have to take immediate action to go back to the original document. They are no less despised, however.

[‡] For example, a call to *window.open(...)* would not generate an exception. The return value in such a case is not standardized, however, making it difficult to detect a blocked pop-up reliably. In Internet Explorer and Firefox, the function will return *null*; in Safari, it will return another special value, *undefined*; in Opera, a dummy window handle will be supplied; and in Chrome, the returned window handle will even have a quasi-functional DOM.

example, in the case of JavaScript, the ability to call *window.open(...)* would be granted to code executed in response to an *onclick* event and revoked shortly thereafter. (In Internet Explorer and WebKit, this permission expires the moment the event handler is exited. Other browsers may recognize a short grace period of one second or so.)

The pop-up blocking feature initially curtailed pop-up advertising but, ultimately, proved to be fairly ineffective: Many websites would simply wait for the user to click anywhere on the page (in order to follow a link or even scroll the document) and spawn new windows in response. Others simply moved on to even more disruptive practices such as interstitials—full-page advertisements you need to click through to get to the content you actually want to read.

The advertising arms race aside, the war on *window.open(...)* is also interesting from the DoS perspective. Creating hundreds of thousands of windows, thereby exhausting OS-enforced limits on the number of UI handles, is a sure way to crash the browser and to disrupt other applications as well. Any mechanism that limits this capability would be, at least in theory, a valuable defense. No such luck: Unbelievably, only Internet Explorer and Chrome sensibly limit the actual number of times *window.open(...)* can be called in response to a single click. In other browsers, once the temporary permission to open windows is granted, the attacker can go completely nuts and open as many windows as she desires.

Dialog Use Restrictions

Window-related woes aside, all web-originating scripts can open certain browser- or OS-handled dialogs. The usefulness of these dialogs to modern web applications is minimal, but they still constitute another interesting part of the browser security landscape. Dialog-initiating APIs include *window.alert(...)*, used to display simple text messages; *window.prompt(...)* and *window.confirm(...)*, used to request basic user input; and *window.print(...)*, which brings up the OS-level printing dialog. A couple of obscure vendor extensions, such as Mozilla's *window.sidebar.addPanel(...)* and *window.sidebar.addSearchEngine(...)* (to create bookmarks and register new search providers, respectively), are also on this list.

The aforementioned JavaScript methods aside, several types of dialogs can be spawned indirectly. For example, it is possible to invoke the *click()* method on a file upload button or to navigate to a downloadable file, which usually brings up the OS-supplied file selection dialog. Navigating to a URL that requires HTTP authentication will also typically bring up a browser-level prompt.

So, what makes dialogs so interesting? The challenge with these prompts is quite different from that of programmatically created windows. Unlike the largely asynchronous *window.open(...)* API, dialogs pause the execution of JavaScript and defer many other actions (such as navigation or event delivery), effectively preventing dialogs from being created in large numbers to exhaust resources and crash the application. But their modal behavior is also their curse: They prevent any interaction with some portion of the browser until the user dismisses the dialog itself.

This creates an interesting loophole. If a new dialog is opened immediately after the old one is closed, the victim may be locked out of a vital portion of the browser UI, often even losing the ability to close the window or navigate away from the offending page. Malware authors sometimes abuse that quirk to force casual, panicked users to perform a dangerous action (such as downloading and executing an untrusted executable) just to be permitted to continue their work: Making any other choice in the script-initiated security prompt will only make the same dialog reappear over and over again.

Probably because of this malware-related tangent, browser vendors have begun experimenting with less disruptive prompting methods. In Chrome, for example, some of the most common modal dialogs have a checkbox that allows the user to suppress future attempts by the page to use the offending API (until the next reload, that is). In Opera, it is possible to stop the execution of scripts on the page. And in both Opera and recent versions of Firefox, many common dialogs are modal only in relation to the document-controlled area of the window, still allowing the tab to be closed or a different URL to be entered in the address bar. Nevertheless, the coverage of such improvements is limited.

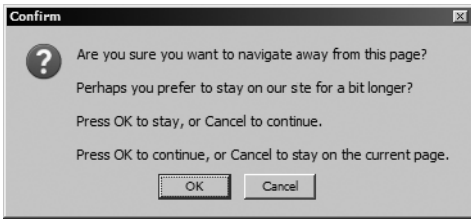


Figure 14-1: Firefox generated a profoundly confusing and vague prompt following the execution of an onbeforeunload *handler on a web page. The handler gives page authors a chance to explain the consequences of navigating away from their page (such as losing any unsaved data) and requests a final decision from the user.[*] In this screenshot, the first and the last line come from the browser itself; the middle two lines are an "explanation" supplied by an (unnamed!) rogue website instead. The security impact of this particular dialog is minimal, but it is a remarkable example of poor UI design. Sadly, a nearly identical dialog is also used by Internet Explorer, and most other browser dialogs are not much better.*

NOTE *Many browser-level dialogs do a poor job of explaining where the prompt originated and its intended purpose. In some cases, such as the Firefox dialog shown in Figure 14-1, the result can be comical—and there is a more sinister side to such goofiness, too. Spawning authoritative-sounding dialogs that claim to be coming from the operating system itself is a common trick used by malware authors to confuse less experienced users. It's not hard to imagine why that works.*

Window-Positioning and Appearance Problems

All right, all right—let's move beyond the arguably uninspiring and unpopular topic of DoS flaws. There is a lot more to the various UI-related APIs—and *window.open(...)* is a particularly curious case. Recall from the discussion earlier in this chapter that this humble function permits web applications not only to create new windows but also to position them in a specific spot on the

[*] For usability reasons, random pages on the Internet are no longer permitted to abort pending navigation by means other than this specific *onbeforeunload* dialog. (Surprisingly, the by-design ability to trap the user on a rogue page forever and cancel any navigation attempts wasn't received well.)

screen. Several other methods, such as *window.moveTo(…)*, *window.resizeTo(…)*, *window.focus()*, or *window.blur()*, further permit such a window to be moved around the screen, scaled, or stacked in a particular way. Finally, *window.close()* allows it to be discreetly disposed of when the script no longer needs it.

As with most other UI-manipulation features, these APIs soon proved to be a source of pain. Following a series of amusing hacks that involved creating "hidden" windows by placing them partly or completely off-screen or by making them really tiny, these functions now require newly created windows to have certain minimal dimensions and to stay entirely within the visible desktop area. (It is still possible to create a window that constantly hops around the screen and evades all mouse-driven attempts to close it, but given what you've read so far, this deserves nothing but a heavy sigh.)

The restrictions on window size do not mean that the entire contents of the address bar have to be visible to the user, however. An undersized window could be leveraged to mislead the user as to the origin of a document simply by carefully truncating the hostname, as shown in Figure 14-2. Browser vendors have been aware of this problem since at least my report in 2010,[1] but as of this writing, only Internet Explorer uses a somewhat convincing if subtle mitigation: It appends "..." at the end of any elided hostnames in the address bar.

Figure 14-2: A window carefully sized by a script so that the real origin of the displayed content is elided in a confusing way. The actual URL of this cat-themed page is http://www.example.com.coredump.cx/*, not* http://www.example.com/.

Another interesting issue with script-controlled window positioning is the prospect of creating several cleverly aligned, overlapping windows to form what appears to be a single document window with an address bar that doesn't correspond to portions of the document displayed. This attack, which I like to call *window splicing*, is perhaps best illustrated in Figure 14-3.

Window positioning offers some interesting if far-fetched attack scenarios, but manipulating the contents of a programmatically created window is also of some relevance to browser security. We have already mentioned that one of the features of the *window.open(…)* API is its ability to hide certain elements of the browser chrome (scrollbars, menus, and so on) in the newly opened window. An example of such a UI-restricting call is

```
window.open("http://example.com/", "_blank", "location=no,menubar=no");
```

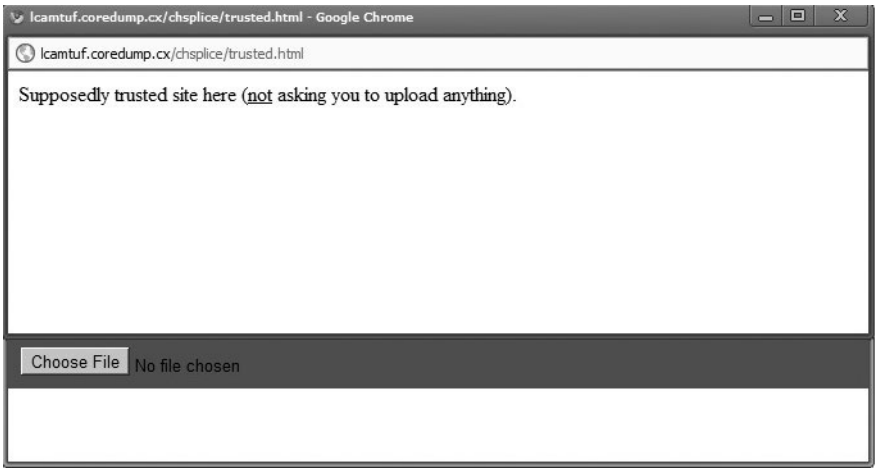


Figure 14-3: A window-splicing attack in Chrome. What may appear as a single document is actually a composite of two overlapping, aligned windows. The user is led to believe that the file upload button comes from the domain shown in the address bar of the top window, but it does not. Certain visual cues indicate foul play (for example, part of the window border has a slightly different hue), but they are too subtle to be easily noticed by the user.

One of these settings, `location=no`, was meant to hide the address bar. This is, of course, a horrible idea: It enables the attacker not only to hide the actual address bar but also to load a page that simply provides a pixel-perfect image of the address bar showing a completely unrelated URL. Heck, with some minimal effort, that fake address bar may even be fully interactive.

Realizing the dangers of this design, most browsers eventually began displaying a minimalistic, read-only address bar in any windows opened with `location=no`; Apple, however, sees no harm in allowing this setting to work as originally envisioned in the 1990s. Too bad: Figure 14-4 shows a simple attack on its UI. (I contacted Apple about this attack sometime in 2010 but have yet to hear back.)

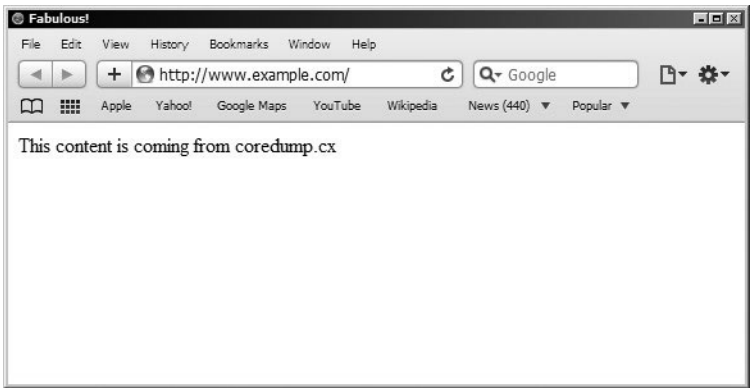


Figure 14-4: Allowing websites to hide the address bar in Safari is a bad idea. The displayed document is not retrieved from http://www.example.com/*. Instead, the page simply displays a screenshot of a real address bar in a window created by* window.open("http://coredump.cx/...", "location=no").

Microsoft has not fared much better: Although they patched up *window.open(...)*, they forgot about *window.createPopup(...)*, an ancient and obscure API still not subject to the necessary checks.

Timing Attacks on User Interfaces

The problems we've discussed so far in this chapter may be hard to fix, but at least in principle, the solutions are not out of reach. Still, here's a preposterous question: Could the current model of web scripting be fundamentally incompatible with the way human beings work? By that, I do not mean merely the dangers of web-delivered social engineering that targets the inattentive and the easily confused; rather, I'm asking if it's possible for scripts to consistently outsmart alert and knowledgeable victims simply due to the inherent limitations of human cognition?

The question is outlandish enough not to be asked often, yet the answer may be yes. Consider that in a typical, attentive human subject, the usual latency between a visual stimulus and a voluntary motor response is between 100 and 300 milliseconds.[2] Humans do not pause for that long to assess the situation after every minute muscle movement; instead, we subconsciously schedule a series of learned motor actions well in advance and process any sensory feedback as it arrives later on. For a split second, we cannot abort a premeditated action, even if something goes horribly wrong.

Alas, on today's personal computers, a lot can happen in as little as one-tenth of that interval. In particular, scripts can open new windows, move them around, or close any existing ones; they can also initiate or abort system-level prompts. In such a setting, designing security-sensitive UIs is not nearly as simple as it seems, and some types of attacks may be simply impossible to defend against without a major paradigm shift in how we design software.

To illustrate the issue, consider a page that attempts to start an unsolicited download of a dangerous file type. The download will typically initiate a browser-level dialog with three options: "open," "save," and "cancel." Sane users will make that last choice—but not if the attacker robs them of a chance to do so.

Let's assume that just milliseconds after the dialog is opened, and perhaps before the user even registers its presence, a new window is created on top that hides it from view. In that window, the attacker plants a carefully positioned button or link that the user is likely to click, for example, a button to dismiss an annoying interstitial advertisement. As the user attempts to perform this perfectly reasonable action, the rogue page may use *onmousemove* events to monitor the position and velocity of the mouse pointer and fairly accurately predict the timing of an upcoming click. Closing the overlay window several milliseconds before that click, only to reveal the "open" button in the same position, will lead the user inevitably to make that choice in the security prompt. There is simply nothing the user can do. (I demonstrated a practical attack on Firefox along these lines in 2007.)[3]

In response to the attacks on security dialogs, a variety of security delays have been implemented in the past few years, requiring anywhere from 500 milliseconds to 5 seconds between the dialog coming into focus and any dangerous buttons being enabled for user input. But such delays do

not sit well with browser UI designers: They hate them, feeling that the product should be as responsive as possible and that annoying the user with non-clickable buttons or countdowns is a significant usability issue. Some have even pushed to remove existing timeouts from legacy UIs.* HTML5 geolocation-sharing prompts are impacted by this view. Many browsers are not protected against the attack on this UI in any significant way.[4]

To further complicate the picture, browser-level user interfaces are not the only concern for UI-timing attacks. The security- or privacy-sensitive functionality of many trusted websites can also be attacked, and fixing that problem is a lot harder than adding delay timers on a handful of known dangerous system-level UIs.

NOTE *Millisecond-level click or keypress hijacking aside, it has been repeatedly demonstrated that with minimal and seemingly innocuous conditioning, healthy and focused test subjects can be reliably tricked into ignoring even very prominent and unusual visual stimuli. The infamous Invisible Gorilla experiment,[5] shown in Figure 14-5, is a particularly well-known example of this. Almost all viewers watching a clip prepared by the researchers fail to notice a plainly visible gorilla in a crowd. The corollary is that even savvy users can be conditioned to ignore cues such as changes to the address bar or to SSL indicators in the browser—a very disconcerting thought. The only reason why we are not trying to solve this problem today is that few exploit writers are behavioral scientists. But if you are a high-profile target, this seems like a risky bet.*

*Figure 14-5: A single frame from the Invisible Gorilla experiment, courtesy of Daniel Simons[6] (*http://dansimons.com/*). When asked to view this video and count the number of times the players pass the basketball, most viewers fail to notice a person in a gorilla suit casually strolling across the room halfway through the clip. Really! Go to* http://theinvisiblegorilla.com/videos.html *and try it on a friend.*

* See, for example, Mozilla bug 561177, where one of the Firefox UI engineers proposed the removal of a security delay from the plug-in installation prompt.

Security Engineering Cheat Sheet

When Permitting User-Created <iframe> Gadgets on Your Site

☑ Don't do so unless you are prepared to live with the consequences. You can't reliably prevent a malicious gadget from launching DoS attacks on your users. Any such gadget will also be able to bring up various obscure dialogs that, as a rule, will not distinguish between your top-level page and the domain the gadget is hosted in.

When Building Security-Sensitive UIs

☑ Because of the risk of UI race conditions, avoid situations where a vital setting can be changed with a single keypress or a single click. Require at least two operations (such as selecting a checkbox and then clicking Save). If single-click actions are unavoidable, consider examining other signals. For example, was the mouse pointer in the current window 500 milliseconds ago?

15

EXTRINSIC SITE PRIVILEGES

To wrap up the discussion of all the noteworthy browser security features, we'll look at a handful of mechanisms that grant special privileges to sites hand-picked by the user or hardcoded by the authors of the browser itself. The approach taken in these cases is in stark contrast to the schemes we have discussed previously, all of which rely on a fairly sensible examination of intrinsic properties of the displayed content. Normally, the implementation would have us look at the source of the document, the context it is displayed in, or the nature of the operation that the document is attempting to perform, but barring the outcome of these checks, the browser would never give preferential treatment to a single otherwise unremarkable origin.

Per-site privileges violate this principle of impartiality in a fairly brutal way, for reasons ranging from questionable to—more commonly—just utilitarian. There are compelling usability reasons to bring certain inherently dangerous features to the browser world, but there is no good way to

programmatically decide which web applications are trustworthy enough to be given access to them. Delegating this task to a human being may be the best thing we can do.*

Naturally, the creation of a caste of privileged applications can be very problematic because the boundaries between any two web applications are not particularly well defined to begin with, making it difficult to contain the permissions precisely. And because the already imperfect boundaries apply only to certain cross-site interactions, vulnerabilities such as XSS or XSRF may further contribute to the misery. In the end, a significant disconnect may develop between the intent of a per-site permission and the actual consequences of such a grant.

Browser- and Plug-in-Managed Site Permissions

When balancing security, privacy, and usability, browser vendors sometimes find themselves between a rock and a hard place. Some proposed features seem essential to the continued growth of the Web but are simply too dangerous to be made available to every website on the Internet. Examples of such problematic mechanisms include giving access to video camera or microphone feeds,† allowing websites to query for user geolocation data,‡ installing browser extensions or themes, or opening desktop notifications.

As a work-around for this problem, vendors require the user to approve the application's request in order for it to be allowed to access a privileged API. On the first attempt to use restricted functionality, the user is typically provided with a visual cue (ranging from an icon to a modal prompt) and given three choices: ignore the request, permit it once, or permanently authorize the requesting site to access the API. Of these choices, the last one is the most interesting: If selected, all future access from a matching host will be automatically approved, sometimes without any further visual indication.

NOTE *Most whitelists look only at the hostname, and not at the protocol or port. Any entry on these lists will therefore match more than one SOP origin. In particular, authorizing* https://fuzzybunnies.com/ *to access your camera may also authorize the non-encrypted site at* http://fuzzybunnies.com/ *to do the same.*

Granting websites access to privacy- or security-sensitive features should be done with care, because, as noted earlier, the implications of doing so extend beyond merely trusting the authors of the whitelisted application.

* It is fair to complain that browsers do not do much to equip users with affirmative signals about the trustworthiness of a visited site, even though many robust indicators may plausibly be arrived at in an automated way. Blacklist-driven attempts to block known malicious sites exist, but given the negligible cost of registering a new domain (or compromising a random existing one), these approaches are arguably of less value.

† This functionality is currently supported only by plug-ins, such as Adobe Flash, but on track to become a part of HTML5.

‡ This API derives user location from parameters such as the current IP address, the list of nearby wireless networks or cell towers, or the data supplied by a hardware GPS receiver. With the exception of GPS data, it may be necessary to consult an external service provider to map these inputs to physical coordinates.

Permission is granted to any content executed in the matching origin, regardless of how the payload got there, greatly amplifying the impact of simple (and, in the long run, inevitable) implementation bugs. A script injection vulnerability in a privileged origin no longer merely exposes the data stored within the application but may also leak client-originating sensitive data feeds.

Hardcoded Domains

In addition to the list of user-authorized privileged domains, some browsers or browser plug-ins come with a list of vendor-selected sites or SOP origins that are given substantial privileges to reconfigure or update portions of the browser or the operating system. Some of the most prominent examples of this trend include *update.microsoft.com*, which is recognized by ActiveX controls that ship with Microsoft Windows and is allowed to install software updates; *addons.mozilla.org* and *chrome.google.com*, recognized by their corresponding browsers and given special privileges to install extensions or themes; or *www.macromedia.com*, which is allowed to reconfigure Adobe Flash.

The designs of these mechanisms vary and, as a rule, are not documented in a satisfactory way. Some features require second-level verification, such as a cryptographic signature or user consent, but others do not. Broadly speaking, the proliferation of such privileged domains is troubling, because it is clear that they will not be immune to the usual security problems that plague the rest of the modern Web. Case in point: *http://xssed.com/* lists six publicly reported XSS vulnerabilities in *addons.mozilla.org*.[1]

Form-Based Password Managers

Surprised? Don't be. Mentioning password managers may seem out of place, but it is very useful to consider this technology as an indirect form of a site-bound privilege. Before we explain, let's briefly review why password management is implemented in modern browsers to begin with and how it actually operates.

The answer to the first question is fairly simple: Today, almost every major website requires, or at least strongly encourages, all visitors to open an account. Logging in is typically necessary in order to customize the appearance of the site and is a prerequisite for interacting with other registered users. Unfortunately, these site-specific authentication systems are not synchronized (save for several limited-scale "federated login" experiments, such as OpenID),[2] and they effectively force the general population to create and memorize several dozen robust passwords, one for every destination frequented. This approach is difficult to sustain and leads to rampant and dangerous password reuse; that's where browser vendors decided to step in.

Form-based password managers are an inelegant but pragmatic solution to the problem of coping with the proliferation of per-site credentials. They apply simple heuristics to detect the submission of normal-looking login forms (the browser looks for an *<input type=password>* field and then perhaps examines the names of form fields for strings such as *user* and *pass*). When a suitable form is detected, the browser will offer to save the associated login

information in a persistent store on the hard drive,[*] and if the user consents, it will then automatically retrieve and paste this data into matching forms encountered later on. In Firefox, Chrome, and Safari, the process of retrieving a stored password is automatic; in Internet Explorer and Opera, an additional user gesture may be required to confirm the intent.

The design of password managers is fragile but has one clear benefit: It works right away even without official support (or, for that matter, informed consent) from any websites. Web applications that are unhappy about this feature may opt out by appending a poorly named *autocomplete=off* parameter to the offending password field,[†] but beyond that, the process is almost completely seamless.

The primary way that every in-browser password manager protects stored data is by tying the credentials to the SOP origin where they were originally entered—paying close attention to the hostname, protocol, and port. Some browsers also consider secondary indicators, such as the ordering or naming of form fields, the URL path to the form, or the address to which the credentials are sent. (As we know from Chapter 9, such scoping measures are not particularly useful from the security standpoint due to the operation of the same-origin policy.)

In browsers that autocomplete login forms without the need for human interaction, it is sensible to look at the mechanism as a form of a privileged API: Any content executing in the appropriate origin will be able to request browser-stored credentials by constructing a believable-looking form and then waiting for it to be automatically populated with login data. In order to read back this information, the script merely needs to examine the *value* property of the DOM element associated with the password field.

NOTE *Removing the ability to inspect values of password fields may seem like a simple way to improve the scheme, but it is not a very good one. The data could still be stolen by, say, waiting for password autocompletion, changing the data submission method from POST to GET, and then calling* submit() *on the login form. These steps would result in navigation to a page that has the password plainly visible in the* location.search *string. (Plus, many web applications have legitimate uses for reading back these fields on the client side, for example, to advise on password strength.)*

As should be clear, the most serious risk associated with password managers is the amplification of XSS bugs. In web applications that use *httponly* cookies, a successful exploitation of an XSS flaw may give the attacker only transient access to a user's account, but if the same vulnerability can be leveraged to steal a user's password, the consequences are more dire and longer-lived.[‡]

[*] This data may be stored on disk as a plaintext representation, a naïvely obfuscated string, or a properly encrypted value protected with a "master" password that needs to be entered beforehand. All three methods are comparably vulnerable to determined attackers with access to the local system, but the plaintext approach is sometimes frowned upon, as it is more exposed to nosy but nontechnical users.

[†] Despite the name, this stops the browser from recording the password and not just from autocompleting it.

[‡] Such consequences may extend beyond the affected application: Even with password managers in place, password reuse is a common, unfortunate trend.

More obscure side effects are possible, too. For example, any application that allows users to construct custom form-based surveys must carefully restrict the layout of the generated forms or risk doubling as a password-harvesting tool.

Internet Explorer's Zone Model

Internet Explorer's zone model[3] is a proprietary attempt to reconcile the different security requirements that users (or system administrators) may have for different types of web applications, for example, a banking page and an online game. Microsoft's approach is to establish several predefined classes of websites—known as *zones*—each with its own set of configurable security permissions. The five supported zones are these:

- **My computer (aka local machine)** This hidden zone is used for all local *file:* resources (with one exception—more about it soon). The user cannot add or remove any elements from this set and cannot change its security settings through the normal user interface. Administrators and developers can modify the registry or use *urlmon.dll* hooks to override settings, however.
- **Local intranet** This zone is meant to include trusted applications on a user's local network. By default, *local intranet* enjoys many problematic privileges, such as unrestricted access to the system clipboard, the ability to open windows without an address bar, or the ability to bypass the usual frame navigation security checks (the descendant policy, outlined in Chapter 11). Members of this set are detected automatically using several configurable heuristics, and they may include destinations with non–fully qualified hostnames, addresses on the HTTP proxy exemption list,[*] or remote *file:* URLs accessed over SMB. Manual inclusion of sites in this zone is also possible (in addition to or instead of the built-in heuristics).

  NOTE *The local intranet zone makes an implicit connection between a local network and a trusted environment. This connection is often dubious in the modern-day environment, especially given the prevalence of public Internet access over unencrypted Wi-Fi: Other uses of the network are not any more trustworthy than a random website hosted across the globe.*

- **Trusted sites** These are nominally empty zones roughly equivalent to *local intranet* in terms of their security settings but managed solely by the user. Autodetection heuristics are unavailable, and all entries have to be created by hand.
- **Restricted sites** In these nominally empty zones, the user may add "untrusted" destinations. The default settings for these zones remove many rudimentary and generally harmless capabilities from the loaded content (for example, *Refresh* headers will not work) while offering limited security benefits.

[*] In configurations where a proxy is required to access protected internal systems but not required to access the Internet, these may have the unintended and scary effect of classifying the entire Web as a local network.

The practicality of this zone seems unclear. Because of the need to whitelist every untrusted site, the zone obviously can't be relied upon as an alternative to browsing the Internet with sensible default settings for previously unseen destinations.

- **Internet** This is a default zone for sites not included in any of the remaining categories. Its default settings match the general browser security model baseline discussed previously in this book.

The concept of zones, coupled with some of their security controls, seems to be a step in the right direction. For example, it allows system administrators to fine-tune the permissions for *file:* documents without affecting the security or convenience of normal browsing—or to prohibit Internet sites from navigating to local, corporate systems (using the setting named "Websites in less privileged web content zone can navigate into this zone"). Unfortunately, the actual implementation of the zone model is muddied by a lack of focus, and in practice, it is misused more often than it is genuinely benefited from.

The first problem evident to anyone trying to master the zone mechanism is its obtuse terminology and the almost-comical complexity of many of the settings. Every zone comes with over 100 checkboxes; some of these will alter the browser security model profoundly, while others have no security consequences whatsoever. (The aforementioned *Refresh* setting is one example of a security no-op; the ability to disable form submission is another.) These two classes of settings are not distinguished in any clear way, and many are nearly impossible to comprehend at a glance. For example, the option "Binary and script behaviors" can be set to "enable" or "disable," but the help subsystem offers no information about what either setting will actually do. The only explanation is provided in the official developer documentation posted on Microsoft's site—but even this document can confuse.[4] See for yourself:

> Internet Explorer contains dynamic binary behaviors: components that encapsulate specific functionality for HTML elements to which they were attached. These binary behaviors are not controlled by any Internet Explorer security setting, allowing them to work on Web pages in the Restricted Sites zone. In Windows Server 2003 Service Pack 1, there is a new Internet Explorer security setting for binary behaviors. This new setting disables binary behaviors in the Restricted Sites zone by default. In combination with the Local Machine Lockdown security feature, it also requires administrative approval for binary behaviors to run in the Local Machine zone by default. This new binary behaviors security setting provides a general mitigation to vulnerabilities in Internet Explorer binary behaviors.

There are many similar cases of settings that require a substantial effort to understand. For example, it is unlikely that even the most seasoned administrators will understand the implications of tweaking settings named "Access data sources across domains" or "Navigate windows and frames across different domains". All this confusion has an interesting consequence: Trusted parties unintentionally dispense dubious advice. For example, Charles Schwab, a prominent investment bank, tells customers to disable the frame navigation

descendant model,[5] essentially making HTML frames unsafe to use not only for Charles Schwab but also for any other website. One of the sites maintained by the Internal Revenue Service provides the same, extremely inconsiderate tip.[6]

The complexity and poor documentation of Internet Explorer's zone settings aside, the other problem with the zone model is the clustering of unrelated permissions. The settings for *local intranet* and *trusted sites* containers enable a random collection of features that may be required by some trusted sites—but none of the trusted sites could possibly require *all* of the permissions the zone entails. Because of this design, adding sites to privileged zones can once more have unexpectedly far-ranging consequences in the case of, say, a trivial XSS flaw.

Mark of the Web and Zone.Identifier

To maintain the integrity of the zone model on downloaded files, Internet Explorer further utilizes two overlapping mechanisms to track the original zone information for any externally retrieved document:

- **Mark of the Web (MotW)** This simple pseudo-HTML tag is inserted at the beginning of HTML documents downloaded via Internet Explorer to indicate their initial source.[7] One example of a MotW tag may be *<!– saved from url=(0024)http://fuzzybunnies.com/ –>*. The URL recorded in this tag is mapped to an appropriate zone; the document is then opened in a unique origin in that zone. The most important consesequence is that the downloaded content is isolated from other *file:* URLs.

  NOTE *The inline nature of MotW is one of its flaws. Faux tags can be pre-inserted by rogue parties into HTML documents downloaded through non–Internet Explorer browsers, saved from email clients, or downloaded by Internet Explorer with a non-HTML extension (and then subjected to content sniffing). Though, to be fair, the privileges of* file*: documents saved without any MotW tags are significant enough to keep attackers relatively uninterested in hopping from the* My Computer *zone to, say,* Local Intranet.

- **Alternate Data Stream (ADS) Zone Identifier** This is a piece of NTFS metadata attached by Internet Explorer (and Chrome) to every downloaded file, indicating the numerical code of the zone the file was retrieved from.[8] The *Zone.Identifier* mechanism is less portable than MotW, and the information is lost when files are saved to non-NTFS filesystems. However, it is also more versatile, as it can be applied to non-HTML documents.

  Zone.Identifier metadata is recognized by Internet Explorer itself, by the Windows GUI shell, and by some other Microsoft products, but third-party software almost universally ignores it. Where it is supported, it may result in a more restrictive security policy being applied to the document; more commonly, it just pops up a security warning about the unspecified risks of opening Internet-originating data.

Security Engineering Cheat Sheet

When Requesting Elevated Permissions from Within a Web Application

Keep in mind that requesting access to geolocation data, video or microphone feeds, and other privileged APIs comes with responsibility. If your site is prone to XSS vulnerabilities, you are gambling not only with the data stored in the application but with the privacy of your users. Plan accordingly and compartmentalize the privileged functionality well. Never ask your users to lower their Internet Explorer security settings to accommodate your application, and do not blindly follow this advice when given by others—no matter who they are.

When Writing Plug-ins or Extensions That Recognize Privileged Origins

You are putting your users at elevated risk due to inevitable web application security bugs. Design APIs robustly and try to use secondary security measures, such as cryptography, to further secure communications with your server. Do not whitelist nonencrypted origins, as they are prone to spoofing on open wireless networks.

PART III

A GLIMPSE OF THINGS TO COME

Following nearly a decade of stagnation, the world of browsers is once more a raging battlefield. In a manner all too reminiscent of the First Browser Wars in the late 1990s, vendors compete by bringing new features to market monthly. The main difference is that security is now seen as a clear selling point.

Of course, objectively measuring the robustness of any sufficiently complex piece of software is an unsolved problem in computing, doubly so if your codebase happens to carry almost two decades worth of bloat. Therefore, much of the competitive effort goes into inventing and then rapidly deploying new security-themed additions, often with little consideration for how well they actually solve the problem they were supposed to address.

In the meantime, standards bodies, mindful of their earlier misadventures, have ditched much of their academic rigor in favor of just letting a dedicated group of contributors tweak the specifications as they see fit. There is talk of making HTML5 the last numbered version of the standard and transitioning to a living document that changes every day—often radically. The relaxation

of the requirement has helped keep ongoing much of the work around W3C and WHATWG, but it has also undermined some of the benefits of having a central organization to begin with. Many recent proposals gravitate toward quick, narrowly scoped hacks that do not even try to form a consistent and well-integrated framework. When this happens, no robust feedback mechanism is in place to allow external experts to review reasonably stable specifications and voice concerns before any implementation work takes place. The only way to stay on top of the changes is to immerse oneself in the day-to-day dynamics of the working group.

It is difficult to say if this new approach to standardization is a bad thing. In fact, its benefits may easily outweigh any of the speculative risks; for one, we now have a chance at a standard that is reasonably close to what browsers actually do. Nevertheless, the results of this frantic and largely unsupervised process can be unpredictable, and they require the security community to be very alert.

In this spirit, the last part of the book will explore some of the more plausible and advanced proposals that may shape the future of the Web . . . or that may just as likely end up in the dustbin of history a few years from now.

16

NEW AND UPCOMING SECURITY FEATURES

You will soon find out that there is little rhyme and reason to how all the new browser features mesh, but we still need to organize the discussion in some way. Perhaps the best approach is to look at their intended purposes and begin with all the mechanisms created specifically to tweak the Web's security model for a well-defined gain.

The dream of inventing a brand-new browser security model is strong within the community, but it is always followed by the realization that it would require rebuilding the entire Web. Therefore, much of the practical work focuses on more humble extensions to the existing approach, necessarily increasing the complexity of the security-critical sections of the browser codebase. This complexity is unwelcome, but its proponents invariably see it as justified, whether because they aim to mitigate a class of vulnerabilities,

build a stopgap for some other hard problem that nobody wants to tackle right now,* or simply enable new types of applications to be built in the future. All these benefits usually trump the vague risk.

Security Model Extension Frameworks

Some of the most successful security enhancements proposed in the past few years boil down to adding flexibility to the original constraints imposed by the same-origin policy and its friends. For example, one formerly experimental proposal that has now crossed into the mainstream is the *postMessage(…)* API for communicating across origins, discussed in Chapter 9. Surprisingly, the act of relaxing SOP checks in certain carefully chosen scenarios is more intuitive and less likely to cause problems than locking the policy down. So, to begin on a lighter note, we'll focus on this class of frameworks first.

Cross-Domain Requests

Under the original constraints of the same-origin policy, scripts associated with one origin have no clean and secure way to communicate with client-side scripts executing in any other origin and no safe way to retrieve potentially useful data from a willing third-party server.

Web developers have long complained about these constraints, and in recent years, browser vendors have begun to listen to their demands. As you recall, the more pressing task of arranging client-side communications between scripts was solved with *postMessage(…)*. The client-to-server scenario was found to be less urgent and still awaits a canonical solution, but there has been some progress to report.

The most successful attempt to create a method for retrieving documents from non-same-origin servers began in 2005. Under the auspices of W3C, several developers working on VoiceXML, an obscure document format for building Interactive Voice Response (IVR) systems, drafted a proposal for *Cross-Origin Resource Sharing (CORS)*.[1] Between 2007 and 2009, their awkward, XML-based design gradually morphed into a much simpler and more widely useful scheme, which relied on HTTP header–level signaling to communicate consent to cross-origin content retrieval using a natural extension of the *XMLHttpRequest* API.

CORS Request Types

As specified today, CORS relies on differentiating between two types of calls to the *XMLHttpRequest* API. When the site attempts to load a cross-origin document through the API, the browser first needs to distinguish between *simple requests*, where the resulting HTTP traffic is deemed close enough to what

* Malicious URL blacklists, a feature supported by (and usually enabled in) all modern browsers, are a prime example of this trend. The blacklist is a lightweight, crude substitute for an antivirus, which is, in turn, a poor substitute for up-to-date and well-designed software. Antimalware features do not make individual attacks any more difficult; they are simply meant to stop the large-scale distribution of unsophisticated malware, based on the assumption that most users are not interesting enough to be specifically targeted or attacked with something clever.

can be generated through other, existing methods of navigation, and *non-simple requests*, which encompass everything else. The operation of these two classes of requests vary significantly, as we'll see.

The current specification says that simple requests must have a method of GET, POST, or HEAD. Additionally, if any custom headers are specified by the caller, they must belong to the following set:

- *Cache-Control*
- *Content-Language*
- *Content-Type*
- *Expires*
- *Last-Modified*
- *Pragma*

Today, browsers that support CORS simply do not allow methods other than GET, POST, and HEAD. At the same time, they ignore the recommended whitelist of headers, unconditionally demoting any requests with custom header values to non-simple status. The implementation in WebKit also considers any payload-bearing requests to be non-simple. (It is not clear whether this is an intentional design decision or a bug.)

Security Checks for Simple Requests

The CORS specification allows simple requests to be submitted to the destination server immediately, without attempting to confirm whether the destination is willing to engage in cross-domain communications to begin with. This decision is based on the fact that the attacker may initiate fairly similar cookie-authenticated traffic by other means (for example, by automatically submitting a form) and, therefore, that there is no point in introducing an additional handshake specifically for CORS.[*]

The crucial security check is carried out only after the response is retrieved from the server: The data is revealed to the caller through the *XMLHttpRequest* API only if the response includes a suitable, well-formed *Access-Control-Allow-Origin* header. To assist the server, the original request will include a mandatory *Origin* header, specifying the origin associated with the calling script.

To illustrate this behavior, consider the following cross-domain *XMLHttpRequest* call performed from *http://www.bunnyoutlet.com/*:

```
var x = XMLHttpRequest();
x.open('GET', 'http://fuzzybunnies.com/get_message.php?id=42', false);
x.send(null);
```

[*] That assumption is not completely correct. For example, prior to the introduction of this scheme, attackers would not have been able to initiate a cross-domain request completely indistinguishable from the submission of a file upload form, but under CORS, such forgery is possible.

The result will be an HTTP request that looks roughly like this:

```
GET /get_message.php?id=42 HTTP/1.0
Host: fuzzybunnies.com
Cookie: FUZZYBUNNIES_SESSION_ID=EA7E8167CE8B6AD93D43AC5AA869A920
Origin: http://www.bunnyoutlet.com
```

To indicate that the response should be readable across domains, the server needs to respond with

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin: http://www.bunnyoutlet.com

The secret message is: "It's a cold day for pontooning."
```

NOTE *It is possible to use a wildcard ("*") in* Access-Control-Allow-Origin*, but do so with care. It is certainly unwise to indiscriminately set* Access-Control-Allow-Origin: * *on all HTTP responses, because this step largely eliminates any assurances of the same-origin policy in CORS-compliant browsers.*

Non-simple Requests and Preflight

In the early drafts of the CORS protocol, almost all requests were meant to be submitted without first checking to see if the server was actually willing to accept them. Unfortunately, this design undermined an interesting property leveraged by some web applications to prevent cross-site request forgery: Prior to CORS, attackers could not inject arbitrary HTTP headers into cross-domain requests, so the presence of a custom header often served as a proof that the request came from the same origin as the destination and was issued through *XMLHttpRequest*.

Later CORS revisions corrected this problem by requiring a more complicated two-step handshake for requests that did not meet the strict "simple request" criteria outlined in "CORS Request Types" on page 236. The handshake for non-simple requests aims to confirm that the destination server is CORS compliant and that it wants to receive nonstandard traffic from that particular caller. The handshake is implemented by sending a vanilla OPTIONS request ("preflight") to the target URL containing an outline of the parameters of the underlying *XMLHttpRequest* call. The most important information is conveyed to the server in three self-explanatory headers: *Origin*, *Access-Control-Request-Method*, and *Access-Control-Request-Headers*.

This handshake is considered successful only if these parameters are properly acknowledged in the response through the use of *Access-Control-Allow-Origin*, *Access-Control-Allow-Method*, and *Access-Control-Allow-Headers*. Following a correct handshake, the actual request is made. For performance reasons, the result of the preflight check for a particular URL may be cached by the client for a set period of time.

Current Status of CORS

As of this writing, CORS is available only in Firefox and WebKit-based browsers and is notably absent in Opera or Internet Explorer. The most important factor hindering its adoption may be simply that the API is not as critical as *postMessage(…)*, its client-side counterpart, because it can be often replaced by a content-fetching proxy on the server side. But the scheme is also facing three principal, if weak, criticisms, some of which come directly from one of the vendors. Obviously, these criticisms don't help matters.

The first complaint, voiced chiefly by Microsoft developers and echoed by some academics, is that the scheme needlessly abuses ambient authority. They argue that there are very few cases where data shared across domains would need to be tailored based on the credentials available for the destination site. The critics believe that the risks of accidentally leaking sensitive information far outweigh any benefits and that a scheme permitting only nonauthenticated requests to be made would be preferable. In their view, any sites that need a form of authentication should instead rely on explicitly exchanged authentication tokens.*

The other, more pragmatic criticism of CORS is that the scheme is needlessly complicated: It extends an already problematic and error-prone API without clearly explaining the benefits of some of the tweaks. In particular, it is not clear if the added complexity of preflight requests is worth the peripheral benefit of being able to issue cross-domain requests with unorthodox methods or random headers.

The last of the weak complaints hinges on the fact that CORS is susceptible to header injection. Unlike some other recently proposed browser features, such as WebSockets (Chapter 17), CORS does not require the server to echo back an unpredictable challenge string to complete the handshake. Particularly in conjunction with preflight caching, this may worsen the impact of certain header-splitting vulnerabilities in the server-side code.

XDomainRequest

Microsoft's objection to CORS appears to stem from the aforementioned concerns over the use of ambient authority, but it also bears subtle overtones of their dissatisfaction with interactions with W3C. In 2008, Sunava Dutta, a program manager at Microsoft, offered this somewhat cryptic insight:[2]

> During the [Internet Explorer 8] Beta 1 timeframe there were many security based concerns raised for cross domain access of third party data using cross site XMLHttpRequest and the Access Control framework. Since Beta 1, we had the chance to work with other browsers and attendees at a W3C face-to-face meeting to improve the server-side experience and security of the W3C's Access Control framework.

* The same claim can be made about the use of HTTP cookies in any other setting and seems equally futile. It is true that ambient credentials cause problems more frequently than some other forms of explicit authentication would, but they are also a lot more convenient to use and are simply not going away.

Instead of embracing the CORS extensions to *XMLHttpRequest*, Microsoft decided to implement a counterproposal, dubbed *XDomainRequest*.[3] This remarkably simple, new API differs from the variant available in other browsers in that the resulting requests are always anonymous (that is, devoid of any browser-managed credentials) and that it does not allow for any custom HTTP headers or methods to be used.

The use of Microsoft's API is otherwise very similar to *XMLHttpRequest*:

```
var x = new XDomainRequest();
x.open("GET", "http://www.fuzzybunnies.com/get_data.php?id=1234");
x.send();
```

Borrowing from W3C's proposal, the resulting request will bear an *Origin* header, and the response data will be revealed to the caller only if a matching *Access-Control-Allow-Origin* header is present in the response.* Preflight requests and permission caching are not a part of the design.

For all intents and purposes, Microsoft's solution is far more reasonable than CORS: It is simpler, safer, and probably just as functional in all the plausible uses. That said, it is also unpopular. It is supported only in Internet Explorer 8 and up, and owing to W3C backing CORS, others have no reason to embrace *XDomainRequest* anytime soon.

In the meantime, a separate group of researchers have proposed a third solution, again acting under the auspices of W3C. Their design, known as Uniform Messaging Policy (complete with a corresponding *UniformRequest* API),[4] embraces an approach nearly identical to Microsoft's. It is not supported in any existing browser, but there is some talk of unifying it with CORS.

Other Uses of the Origin Header

The *Origin* header is an essential part of CORS, *XDomainRequest*, and UMP, but it actually evolved somewhat independently with other uses in mind. In their 2008 paper, Adam Barth, Collin Jackson, and John C. Mitchell[5] advocated the introduction of a new HTTP header that would offer a more reliable and privacy-conscious alternative to *Referer*. It would also serve as a way to prevent cross-site request vulnerabilities by providing the server with the information needed to identify the SOP-level origin of a request, without disclosing the potentially more sensitive path or query data.

Of course, it was unclear whether the subtle improvement between *Referer* and its proposed successor would actually make a difference for the small but nonnegligible population of users who block that first header on privacy grounds. The proposal consequently ended up in a virtual limbo, not being deployed in any existing browsers but also discouraging others from pursuing other solutions such as XSRF or XSSI.[6] (To be fair, the concept was very recently revived under the new name of *From-Origin* and may not be completely dead yet.)[7]

* The reason for this check, even if the response is not authenticated, is to prevent the use of the browser as a proxy (for example, to crawl internal networks or send out spam).

The fate of the original idea aside, the utility of the *Origin* header in specialized cases such as CORS was pretty clear. Around 2009, this led to Barth submitting an IETF draft specifying the syntax of the header,[8] while shying away from making any statements about when the header should be sent, or what specific security problems it might solve:

> The user agent MAY include an Origin header in any HTTP request.
>
> […]
>
> Whenever a user agent issues an HTTP request from a "privacy-sensitive" context, the user agent MUST send the value "null" in the Origin header.
>
> NOTE: This document does not define the notion of a privacy-sensitive context. Applications that generate HTTP requests can designate contexts as privacy-sensitive to impose restrictions on how user agents generate Origin headers.

The bottom line of this specification is that whatever the decision process is, once the client chooses to provide the header, the value is required to accurately represent the SOP origin from which the request is being made. For example, when a particular operation takes place from *http://www.bunnyoutlet.com:1234/bunny_reports.php*, the transmitted value should be

```
Origin: http://www.bunnyoutlet.com:1234
```

For origins that do not meaningfully map to a protocol-host-port tuple, the browser must send the value of *null* instead.

Despite all of these plans, as of this writing only one browser includes the *Origin* header on non-CORS navigation: WebKit-based implementations send it when submitting HTML forms. Firefox seems to be considering a different approach, but nothing specific seems to have been implemented yet.

Security Model Restriction Frameworks

Designs that extend the bounds of the same-origin policy are fairly simple to understand and typically fail securely. If the proposed change is not accounted for in one of the possible code paths, or is simply not supported in a particular browser, the previously implemented, more restrictive logic will kick in. Compared with this, it is far more dangerous to try to erect new boundaries on top of the existing browser security model. That's because every security-sensitive code path must be tweaked to recognize the new scheme and every browser must comply right away, or unexpected problems will arise.

In this section, we will take a quick look at some of the more accomplished attempts to take this dangerous but potentially rewarding path—and explore where they come apart.

Content Security Policy

Content Security Policy (*CSP*) is an unusually comprehensive security framework first proposed by Brandon Sterne of Mozilla in 2008.[9] The framework was originally envisioned as an all-encompassing way to mitigate the impact of common web vulnerabilities, from XSRF to XSS, and as a tool for website owners to perform a variety of non-security content-policing tasks.

In the years that followed, CSP evolved rapidly, and on several occasions, its scope changed in major ways. (For example, the author quickly abandoned the plan to address XSRF vulnerabilities, delegating the job to the yet unrealized extensions of the *Origin* header.) In fact, as of this writing, the canonical Mozilla specification is being rewritten as a W3C draft,[10] resulting in substantial differences in the implementation shipped in Firefox and the partial support implemented in WebKit by Adam Barth. (Internet Explorer and Opera do not support CSP and have not announced any specific plans to embrace it.)

Primary CSP Directives

At its core, Sterne's design permits site owners to specify per-document policies that constrain the ability of the subject document to perform actions that would normally be permitted under the same-origin policy. For example, CSP may prevent a page from loading any external subresources except for images and restrict image sources to only a set of trusted origins, like so:

```
X-Content-Security-Policy: default-src 'none'; img-src http://*.example.com
```

As should be evident from this example, the policies may be encoded in an HTTP header. Under the W3C draft, it is also possible to embed them in the document itself (using <*meta*> tags) or host the policy at an external URL and point to it with *policy-uri*.

For every content source directive, the author of the policy may specify any number of fully qualified origins or wildcard expressions that match multiple hosts, protocols, or ports. Three special keywords (*none*, *self*, and *data:*) correspond to an empty set, the origin associated with the policy-bearing page, or all inline *data:* URLs, in corresponding order.

As of today, the following behaviors can be controlled with CSP directives:

- **Script execution** A *script-src* directive can be used to specify the protocol, host, and port for permissible <*script src=...*> URLs. Normally, the CSP disables the ability to embed scripts inline in the document (whether through standalone <*script*> blocks or via event handlers) and of existing scripts to carelessly pass strings to functions such as *eval(...)*, *setTimeout(...)*, *setInterval(...)*, and so on. Because of this, the *script-src* directive is useful for limiting the impact of XSS vulnerabilities: Any markup injected by the attacker will be limited to loading scripts legitimately hosted in one of the approved origins.*

* CSP offers several ways to shoot yourself in the foot here. For one, it is possible to re-enable script execution with settings such as *inline-script* (Mozilla's naming, changed to *disable-xss-protection* in W3C draft) or *eval-script*. Perhaps less obviously, it is also possible to make the mistake of permitting *data:* or * as a permissible origin or allowing an HTTP origin on an HTTPS site.

- **Plug-in content** This is controlled through *object-src*. Because plug-ins such as Java or Flash may have unconstrained access to the embedding page, the directive should be considered largely analogous to *script-src*, and the two directives must be restricted in a comparable way to achieve any security benefits.
- **Stylesheets and fonts** This is controlled by *style-src* and *font-src*. Unlike its handling of scripts, CSP originally did not prevent inline <*style*> blocks or *style*= parameters from appearing on the page. Therefore, any attacker exploiting an XSS flaw could dramatically alter the appearance and function of the vulnerable page (or worse),* and these two directives only served nonsecurity goals, with the possible exception of limiting mixed-content bugs. Only moments before the publication of the book, the specifications have been amended to include a more robust approach to CSS.
- **Passive multimedia** Directives such as *img-src* or *media-src* control the ability to embed multimedia content from specific origins. As with the original design of CSS controls, this could not have been considered a security feature. For example, in the case of an XSS bug, CSP would not have prevented the attacker from leveraging stylesheets to draw arbitrary shapes on the vulnerable page or even animating them to some extent.
- **Subframes** The *frame-src* directive specifies the acceptable destinations for any <*iframe*> tags encountered on the page; the policy of the parent page is not inherited by the framed document. To preserve the value of other XSS mitigations, steps must be taken not to allow *data:* URLs here (see Chapter 10).
- **Default policy** Known as *default-src* in the W3C draft, and under a more cryptic name (*allow*) in Mozilla documentation, the directive specifies fallback behavior for any content not covered by a more specific directive. The directive is required, even in cases where it is technically unnecessary.

NOTE *It may be unfortunate that CSP directives are selected to map very closely to individual HTML tags, instead of grouping functionally similar behaviors. Because of this, it is difficult to appreciate the tricky interactions among settings such as* script-src*,* frame-src*, and* object-src*. Also, the approach is simply not very future-safe: There already are some peripheral classes of subresources (such as "favicons") that are excluded from CSP altogether, and that list will probably unintentionally grow.*

In an unusual departure from the subresource-driven model outlined thus far, CSP also features an oddball directive called *frame-ancestors*. This parameter is meant to mitigate the impact of clickjacking by specifying the allowed ancestors for the current document in a manner similar to the better-established *X-Frame-Options* header (outlined in Chapter 11). The *frame-ancestors* logic is completely independent of *default-src* or any other parts of CSP; its default value is "*".

* Remember advanced selectors in CSS3? By cleverly leveraging them in injected stylesheets, some information about the strings appearing on the page may be conveniently relayed to a third-party server without the use of JavaScript.

Many other possible extensions of the policy are being discussed as of this writing. These include a *script-nonce* directive that could be used to more securely embed inline scripts (every script block must begin with a policy-specified, unpredictable token, often making XSS exploitation harder) and a *sandbox* directive, which offers an alternative interface to another security mechanism, discussed in "Sandboxed Frames" on page 245.

Policy Violations

The policy specified according to these rules constrains the behavior of the underlying document. Violations normally result in a failed subresource load, the failure to execute an inline script, or the inhibition of page rendering (in the special case of *frame-ancestors*).

Because CSP controls a wide range of content behaviors, and because the default failure mode is fairly brutal, the authors perceived a need to ease the worries of webmasters. To make CSP more user-friendly, and perhaps also in a naïve attempt to offer exploit detection, an optional feature of CSP allows the browser to report all policy violations immediately back to the owner of the site. This feature can be enabled through the *report-uri* keyword in the policy. To further simplify deployment, it is also possible to roll out any policy—or part thereof—in a "soft" mode, where violations result only in an HTTP notification but do not actually break the page. This is achieved by specifying the policy inside a header named *X-Content-Security-Policy-Report-Only*.[*]

Criticisms of CSP

CSP is a remarkably sensible and consistent design compared to most of the one-off security features proposed or deployed in the browser world. Nevertheless, from its inception, the proposal has been haunted by recurring design and implementation concerns.

Perhaps the most prosaic complaint about CSP is a nonsecurity one: In order to benefit from the XSS defenses offered by the framework, webmasters have to move all inline scripts on the page (often hundreds of individual snippets of code) to a separately requested document; in the new drafts of CSP, the same will be required for all stylesheets. The complexity of retrofitting existing pages to work with CSP and the performance penalty of an additional HTTP request are often prohibitive. (It may be possible to resolve this problem with the *script-nonce* extension proposed in the most recent drafts.)

A more fundamental concern with the design of CSP is that the currently envisioned origin-level granularity of the rulesets may not offer a sufficiently robust defense against XSS. Consider the fact that any complex, real-life domain may well host a dozen largely separate web applications, each consisting of hundreds of possibly unrelated static scripts and JavaScript APIs. Attackers exploiting an XSS vulnerability in a CSP-protected site are prevented from directly executing a malicious script, but they may be able to put the application into an inconsistent and possibly dangerous state by loading the existing

[*] As a side note, this feature is useful not only for short-term experiments but also for detecting noncritical issues on an ongoing basis. For example, the owner of a site may leverage it to detect mixed-content issues by creating a report-only policy for HTTPS pages that will be violated by any HTTP scripts.

scripts in the wrong context or in an incorrect sequence. The history of vulnerabilities in nonweb software suggests that such state corruption conditions are exploitable more often than we may think.

An even more troubling prospect is that an attacker can load a subresource that is not truly a script but that might be mistaken for one. An extreme example of this may be a browser supporting E4X (see Chapter 6): Any valid XHTML document in which the attacker can place a nominally harmless string—say, *{alert("Hi mom!")}*—may result in code execution when loaded via <*script src*=…>. Recognizing this problem, the developers decided to require whitelisted *Content-Type* values for any scripts loaded under CSP, but even this approach is often insufficient.

To understand what may go wrong, consider the exceedingly common practice of hosting public JSONP APIs in which the client can specify the name of the callback function:

```
GET /store_locator_api.cgi?zip=90210&callback=myResultParser HTTP/1.0
...

HTTP/1.0 200 OK
Content-Type: application/x-javascript
...
myResultParser({ "store_name": "Spacely Space Sprockets",
                 "street": … });
```

Such an API anywhere within a CSP-permitted origin may be leveraged by an attacker to call arbitrary existing functions in the client-side code, perhaps together with attacker-controlled parameters. And if the *callback* string is not constrained to alphanumerics (and why should it be?), specifying *callback=alert(1);//* will lead to straightforward code injection.

Issues with granularity aside, CSP deserves some gentle criticism for its sometimes puzzling and detrimental lack of focus. On one hand, through the inclusion of directives such as *frame-descendants* or *sandbox*, it seems to be flirting with the idea of building a single, unifying browser security framework—only to unexpectedly exclude XSRF flaws from its scope without offering a viable alternative beyond a vague mention of *Origin*. On the other hand, the proposal often aspires to be just a "Content Policy," with no special attention paid to offering sufficiently robust and intuitive security properties. The ease of creating dangerous script policies, coupled with the originally ineffective policing of stylesheets and images, is a testament to this trend.

Sandboxed Frames

Sandboxed frames[11] are an extension of the normal <*iframe*> behavior. They allow the owner of the top-level page to place certain additional restrictions on the embedded document along with any of that document's subframes. The goal is to make it safer for web applications to embed potentially

untrusted advertisements, gadgets, or preformatted HTML documents on an otherwise sensitive site. The refinement of the design and the initial implementation of this feature in WebKit (which is currently the only engine supporting it) was driven by Adam Barth.

NOTE *Curiously, sandboxed frames are not exactly a novel idea: Microsoft came up with a similar proposal almost a decade earlier. Since version 6, Internet Explorer has supported a proprietary* security=restricted *parameter, which forces the target frame to be rendered in the Restricted Zone, effectively removing its ability to execute scripts, navigate to other locations, and so on. However, no one seemed interested in using this feature for anything other than bypassing certain client-side JavaScript security mechanisms (most notably, anticlickjacking checks). We will soon know whether the HTML5 successor fares any better.*

The design of sandboxed frames is fairly simple: Any frame embedded in a document may be constrained by specifying the *sandbox* parameter on the appropriate <*iframe*> tag. By default, the document subject to this restriction is prevented from executing scripts and performing certain types of navigations. The permissions may be fine-tuned with one or more whitespace-delimited keywords, specified as a value for the *sandbox* parameter itself:

- ***Allow-scripts*** In the absence of this keyword, the document displayed inside the frame will be unable to execute JavaScript code. The primary function of this feature is to prevent the embedded document from performing DoS attacks, opening browser dialogs, or employing any other complex automation of the page.
- ***Allow-forms*** When this keyword is absent, any HTML forms encountered in the embedded document will not work. This mechanism is designed to prevent the framed content from exploiting its placement on a trusted website to phish for sensitive information. (Note that with *allow-scripts* enabled, there is little or no point in *allow-forms*. Scripts may easily construct form-like controls and automatically relay the collected information to another site without the need for a functioning <*form*> tag.)
- ***Allow-top-navigation*** This keyword re-enables the ability of the embedded page to navigate the top-level window. This type of navigation is normally permitted as one of the exceptions to the same-origin policy (see Chapter 11), and it may be abused simply to prevent the user from interacting with the embedding site or to carry out phishing attacks.
- ***Allow-same-origin*** Without this flag, the content inside a sandboxed frame is assigned a unique, randomly selected, synthetic origin. This prevents the page from accessing any origin-bound content that would normally be available to scripts executing in the domain it is nominally hosted in. The inclusion of *allow-same-origin* removes the synthetic origin and permits same-origin data access.

Scripting, Forms, and Navigation

The first three restrictions available to sandboxed frames—scripting, forms, and navigation—are fairly intuitive and safe to use. Their value is diminished only by the need to also disable all plug-ins whenever the *sandbox* attribute is used, because frameworks such as Flash or Java do not honor the extension and would allow any embedded applets to bypass the newly added browser checks. Unfortunately, the three most obvious use cases for sandboxed frames—embedded advertisements, videos, and games—rely heavily on Flash, thus rendering this security mechanism much less useful than it might otherwise be.

Synthetic Origins

The last mechanism on the list, synthetic origins, is far more problematic and is likely misguided. It is envisioned primarily as a way to make it possible for untrusted documents (such as incoming HTML-based emails in a webmail interface) to be served as is, along with the rest of the application, while preventing these untrusted documents from accessing sensitive data.

Unfortunately, the concept of synthetic origins creates more problems than it solves. For one, unless the URL of the embedded document is unpredictable, the attacker may simply navigate to it directly in a new browser window, in which case the browser will not see the *sandbox* attribute at all.

As an attempt to work around this problem, the authors of the specification eventually proposed the use of a specialized MIME type (*text/html-sandboxed*) for content meant to be shown only in a sandboxed frame. Their reasoning is that browsers will normally not recognize this MIME type and will not display it inline and that a special case may be created in the *<iframe>* handling code. Of course, as should be clear from Chapter 13, such a defense is inadequate, because some browsers and plug-ins will render *text/html-sandboxed* responses inline or interpret the returned data in other troubling ways (say, as *crossdomain.xml*).

The concept of synthetic origins is also highly problematic given the fragmentation of origin- or domain-level security mechanisms in a typical browser. For example, dangerous interactions are possible with password managers, which must be explicitly prevented from autocompleting login forms in the sandboxed documents. Also, special logic must be added to security prompts, such as the one associated with the geolocation API.

After some trial and error, the implementation currently available in WebKit resolved many of these issues on a case-by-case basis. That said, future implementations are likely to fall for this trap repeatedly, especially since the HTML5 specification considers the behavior of these features to be out of scope and does not specify the required behavior in any way.

NOTE *Removing synthetic origins leads to trouble, too: If the user clicks on a same-site link in a sandboxed advertisement and that link opens in a new window, the browser probably should prevent the unrestricted scripts in the new window from traversing the* opener *object to perform actions that its parent is prohibited from performing on its own.*

Strict Transport Security

One of the most significant weaknesses in the design of HTTPS is that users often begin navigation by typing in a protocol-less URL in the address bar (such as *bankofamerica.com* rather than *https://www.bankofamerica.com*), in which case the browser will presume HTTP and send the initial request in plaintext. Even if the site immediately redirects this traffic to HTTPS, any active attacker on the victim's network may intercept and modify that initial response, preventing the user from ever upgrading to a secure protocol. In such case, the absence of a tiny lock icon in the browser UI will be very easy to miss.

This problem, as well as several peripheral issues related to mixed content and cookie scoping, prompted Jeff Hodges and several other researchers to draft a proposal for HTTP Strict Transport Security (HSTS, or STS for short).[12] Their approach (currently supported in WebKit and Firefox) allows any site on the Internet to instruct the browser that all future requests made to a particular hostname or domain should always use HTTPS and that any HTTP traffic should be automatically upgraded and submitted only over HTTPS.

The reasoning behind the design of HSTS is that the user's first interaction with a particular domain is unlikely to occur over a connection that is being actively tampered with—but that, over time, as the user roams on open wireless networks, the chances of encountering an attacker increase rapidly. HSTS is, therefore, an imperfect defense, but in practice it is usually good enough.

The HSTS opt-in header may appear in HTTPS responses, looking something like this:

```
Strict-Transport-Security: max-age=3000000; includeSubDomains
```

NOTE *For HSTS to offer reasonable protection,* max-age *(the number of seconds that the STS record may be stored in the browser) must be set to a value substantially higher than the usual worst-case time between visits to the site. Because there is no easy way to disable or override HSTS when something goes wrong with the HTTPS site, website owners will be tempted to choose a value small enough to minimize disruption when they mess something up and have to revert. It is not clear whether this conflict of interests will lead web programmers to make optimal choices.*

The negative security consequences of this design are fairly unremarkable: There is a slightly elevated risk of DoS attacks, because an attacker could inject this response header into a domain that is not fully HTTPS enabled. There is also the possibility of using a unique combination of HSTS settings for several decoy hostnames to tag a particular instance of a browser, offering yet another alternative to cookie-based user tracking. Neither of these concerns is particularly pronounced, however.

Unfortunately, as with other restriction-adding frameworks discussed in this section of the book, the mechanism sounds great in principle, but it's difficult to fully account for how it may interact with other legacy code. In particular, unless the *includeSubDomains* flag is used, HSTS offers unexpectedly little protection for HTTP cookies: Cookies not marked as *secure* may still be intercepted simply by inventing a nonexistent subdomain and intercepting the HTTP request made to that destination.* (Even *secure* cookies could be clobbered in a similar fashion, just not read back.)

In a similar vein, the enforcement of HSTS on requests originating from plug-in-based content is unlikely to work well.

Private Browsing Modes

Private browsing, colloquially known as the "porn mode," is a nonstandardized feature available in most up-to-date browsers. It is meant to create a nonpersistent browsing sandbox, isolated from the main browser session, which is completely discarded as soon as the last private browsing window is closed. In a sense, this mechanism can be considered a form of content isolation added on top of the existing browser security paradigms, so it seems fitting to briefly mention it now.

With the exception of Chrome, most browser vendors do not accurately explain the security assurances associated with private browsing. Unfortunately, the intuitive understanding of the term is quite different from what browsers can actually deliver.

Arguably, the most straightforward interpretation of the feature is that a private browsing session should be perfectly anonymous and that no data about the user's activity will persist on the system. These two assumptions are already partly undermined by the constraints imposed by the networking stacks and the memory management practices of modern operating systems. But even within the browser itself, the goal of reasonable anonymity is nearly impossible to achieve. Almost every stateful browser mechanism, from geolocation or pop-up permissions to Strict Transport Security to form autocompletion to plug-in-based persistent data storage, must be modified in order to properly account for the distinction between the two browsing modes, and for each vendor, achieving that goal is an uphill battle. Perhaps more frustratingly, anonymity is also undermined by the ability of scripts to uniquely fingerprint any given system simply by examining its characteristics—such as the set of installed plug-ins, fonts, screen resolutions, window sizes, clock drift, or even the behavior of noncryptographically secure PRNGs.[13]

In the end, despite appearances to the contrary, private browsing mode is suitable only for preventing casual data disclosure to other nontechnical users of the same machine, and even that goal is sometimes difficult to achieve.

* Recall from Chapter 9 that host-scoped cookies are fairly tricky to create in some browsers and outright impossible to have in Internet Explorer.

Other Developments

The security features discussed previously in this chapter aim to shift the boundaries between web applications and change the way sites interact with each other. Another group of proposed mechanisms escapes this simple classification yet is important or mature enough to briefly mention here. We'll review some of them now.

In-Browser HTML Sanitizers

XSS vulnerabilities are by far the most common security issue encountered in modern web applications. It must be surprising, then, that so few of the proposed security frameworks aim to address the problem in a comprehensive way. True, CSP is a strong contender, but it requires a radical change in how web applications are written, and it can't be deployed particularly gradually or selectively. Sandboxed frames, on the other hand, are probably too resource-intensive and too awkward to use for the most common task of displaying hundreds of individual, short snippets of user-supplied data.

Perhaps the best solution to many XSS woes would be a method for web frameworks to provide the browser with a parsed, unambiguous, binary DOM tree. Such a solution would eliminate many of the issues associated with template escaping and HTML sanitization. A more down-to-earth alternative might be to equip web developers with a robust tool to mark the boundaries of an attacker-supplied string and restrict the behavior or appearance of the embedded payload without having to escape or sanitize it. One might think of syntax such as this:

```
<sandbox token="random_value12345" settings="allow_static_html">
   ...any unsanitized text or HTML...
</sandbox token="random_value12345">
```

Were such a tool to be used, the attacker would be unable to escape such a sandbox and remove the restriction on scripting without guessing the correct value of the randomly generated *token* boundary.

Sadly, such a proposal is unlikely to become a part of HTML5 or to ship in any browser, because this serialization is fundamentally incompatible with XML, and revising XML itself to allow an obscure use case in HTML is a difficult act to pull off. Depressingly, XML already offers a similar method of encapsulating arbitrary data inside a *<![CDATA[...]]>* block, but absent a token-based guard, this sandbox can be escaped easily when exploiting XSS.

On the flip side, it is considerably easier to restrict the privileges of any HTML generated by scripts on the client side. Beginning with Internet Explorer 8, Microsoft offers a simple and somewhat inflexible *toStaticHTML(...)* API,[14] which promises to remove JavaScript from any fully qualified bit of HTML passed to it as a parameter. The output of this method is designed to be safe to assign to the *innerHTML* property somewhere in the existing DOM.*

* Amusingly, the HTML parser in Internet Explorer is apparently so obtuse that even the authors of *toStaticHTML(...)* had some trouble following it. Since its introduction, the API has suffered from a fair number of bypass vulnerabilities, most frequently related to the handling of CSS data.

Microsoft's proposal is fine, but it dances around the most common and problematic task of safely displaying server-supplied documents. And its API has a minor but entirely unnecessary weakness: It makes it unexpectedly dangerous to trim or concatenate the sanitized *toStaticHTML(...)* output after the call but before the *innerHTML* assignment, a practice that many web developers will probably attempt. A more sensible approach would be to allow content sanitization only upon assignment to *innerHTML*. In fact, WebKit engineers briefly discussed a proposal for such an API (alternately named *innerStaticHTML* or *safeInnerHTML*), but the effort seems to have fizzled out long ago.

XSS Filtering

Reducing the incidence of cross-site vulnerabilities is difficult, and so is limiting their impact. Because of this, some researchers have concluded that detecting and stopping the exploitation of such flaws may be a better choice. And so, around 2008, David Ross of Microsoft announced the inclusion of XSS-detection logic in the upcoming release of Internet Explorer 8;[15] several months later, Adam Barth implemented a similar feature in WebKit. The implementations compare portions of the current URL with any strings appearing on the retrieved page or passed to APIs such as *document.write(...)* and *innerHTML*. If that comparison reveals that a portion of JavaScript present on the page may have originated with an improperly escaped URL parameter, the relevant portion of the page may be substituted with a harmless string.

Sadly, this seemingly elegant idea is known to cause serious problems. Accidental false positives aside (users of Internet Explorer 8 will have unexpected trouble visiting *http://www.google.com/search?q=<script>*), the filter may also be tripped for ill purposes by appending a legitimate portion of the page as a nonfunctional parameter in the URL. In one extreme and now resolved case, this behavior was leveraged to create XSS vectors where none had existed before, simply by tricking the browser into haphazardly rearranging the markup.[16] But more fundamentally, it's risky for any complex web application to selectively disable attacker-selected script blocks, even if the structure of the page is otherwise correctly preserved, and such a tweak may easily put the client-side code in an inconsistent or dangerous state. For example, consider an online document editor that implements each of the following in a separate *<script>* block:

1. Initializes the internal state of the editor and creates the UI with an empty starting document.
2. Loads the current version of the document requested by the user in a URL parameter with error checking to catch any potential network problems.
3. If no errors are detected, enters an interactive editing mode and automatically saves the current state of the document every 30 seconds under the URL-derived ID.

In this not entirely unreasonable design, the ability to remove step two can be disastrous because the next step could overwrite the existing, server-stored document with a blank copy. D'oh.

This problem could have been avoided by using much simpler design whereby any suspected XSS attacks would result in the browser simply refusing to render the document. Alas, the relatively high incidence of accidental false positives prevented the authors from taking this route. Only after some debate did Microsoft decide to offer a "strict" blocking mode on an opt-in basis, toggled by a response header such as this:

```
XSS-Protection: 1; mode=block
```

NOTE *In addition to the risk of false positives, XSS filters are also prone to false negatives, a situation that probably can't be improved by much. By design, these filters will never be able to detect the arguably more dangerous stored XSS vulnerabilities, where incorrectly escaped data comes from a source other than the followed link. But even beyond that, the multitude of (often implicit) input escaping schemes and the growing use of* location.hash *or* pushState *(Chapter 17) as a method to store application state make it difficult to formulate an accurate connection between what the browser sees in the address bar and what the application makes of the received URL.*

Security Engineering Cheat Sheet

Approach experimental browser security features with care, particularly when dealing with mechanisms that create finer-grained security boundaries. Ensure that any application leveraging these mechanisms will degrade safely in a noncompliant browser.

- ☑ ***Cross-domain XMLHttpRequest (CORS):*** Fairly safe, but easy to misuse. Avoid non-simple requests and do not permit arbitrary headers or methods. If you have control over the server-side application framework, consider automatically stripping *Cookie* headers on incoming CORS requests with nonwhitelisted *Origin* values to minimize the risk of accidentally sharing user-specific data. To minimize the incidence of mixed-content bugs, consider rejecting HTTPS *Origin* values on any requests received over plain HTTP.

  Be wary of *Access-Control-Allow-Origin: **, and if you need to use it, make sure it is only returned for the location you intend to share.
- ☑ ***XDomainRequest:*** This is safe to use. As with *XMLHttpRequest*, restricting access to HTTP APIs from HTTPS origins may be a good way to stamp out mixed-content bugs.
- ☑ **Content Security Policy:** This is safe to use as defense in depth. Review the caveats related to the interactions among *script-src*, *object-src*, and so on, and the dangers of permitting *data:* origins. Do not accidentally allow mixed content: Always specify protocols in the rulesets and make sure they match the protocol the requesting page is served over.
- ☑ **Sandboxed frames:** This is safe to use as a way to embed gadgets from other origins, but the mechanism will fail dramatically in noncompliant browsers. You should not sandbox same-origin documents.
- ☑ **Strict Transport Security:** This is safe to use as defense in depth. Be sure to mark all relevant cookies as *secure* and be prepared for the possibility of cookie injection via spoofed, non-STS locations in your domain. Use *includeSubDomains* where feasible to mitigate this risk.
- ☑ ***toStaticHTML(...)***: This is safe to use where available, but it is difficult to substitute on the client side in noncompliant browsers. Bypass vulnerabilities have an above-average chance of recurring in the API due to the design of the filter.
- ☑ **Private browsing:** Do not rely on this mechanism for security purposes.
- ☑ **XSS filtering:** Do not rely on this mechanism for security purposes. Always explicitly specify *XSS-Protection: 1; mode=block* or *XSS-Protection: 0* in HTTP responses. The default is fairly unsafe.

17

OTHER BROWSER MECHANISMS OF NOTE

To conclude the third part of the book, we briefly enumerate some of the recently implemented or simply planned APIs that, although not designed for security purposes, may substantially change the security landscape in the coming years. For example, some change the types of data that web applications have access to or alter the way the browser communicates with the outside world.

The following list is necessarily incomplete: New, reasonably plausible designs are drafted every week, and old approaches are scrapped at a moment's notice, often long before shipping in an actual browser. Still, this chapter should serve as an interesting snapshot of what the future may bring.

URL- and Protocol-Level Proposals

These features seek to change the processes surrounding the behavior of links, the address bar, and the exchange of data over the wire.

Protocol registration

Web applications commonly assume the handling of URL schemes previously reserved for "real" desktop software. One prime example of this may be the *mailto:* protocol, which was originally meant to instantiate a stand-alone mail application but which is often more sensibly routed to webmail interfaces today. To this end, Mozilla proposed and WebKit embraced a simple *navigator.registerProtocolHandler(...)* API.[1] When this API is invoked, the user is presented with a simple security prompt, and if the action is approved, a URL-based handler is associated with a particular scheme. As of today, the associated prompts are vulnerable to the race conditions outlined in Chapter 14, and they seem to be lacking in other ways, as shown in Figure 17-1.

Figure 17-1: A seriously confusing prompt in Firefox. The prompt shown in the upper area of the browser window was generated by the browser in response to a call to the registerProtocolHandler(...) *API, with the protocol name set to "doing really awesome stuff" and application name set to "Firefox (mozilla.org)". This particular example is harmless, but more sinister abuse is within reach.*

Address bar manipulation

The newly introduced HTML5 *history.pushState(...)* API,[2] supported by Firefox, WebKit, and Opera, permits the currently displayed document to change the contents of the address bar to any other same-origin URL, without actually triggering a page transition normally associated with this step. The API offers a superior alternative to the widespread abuse of *location.hash* to store application state. Interestingly, despite its simplicity, it has already led to a fair number of interesting security bugs. For example, some implementations briefly allowed not only the top-level document

but also any dodgy third-party frames to change the top-level URL shown in the address bar, and they permitted origins such as *about:blank* to put largely unconstrained gibberish in the URL field.

Binary HTTP

SPDY[3] ("Speedy") is a simple, encrypted drop-in replacement for HTTP that preserves the protocol's key design principles (including the layout and function of most headers). At the same time, it mini- mizes the overhead associated with delivering concurrent requests or with the parsing of text-based requests and response data. The protocol is currently supported only in Chrome, and other than select Google services, it is not commonly encountered on the Web. It may be coming to Firefox soon, too, however.

HTTP-less networking

WebSocket[4] is a still-evolving API designed for negotiating largely unconstrained, bidirectional TCP streams for when the transactional nature of TCP gets in the way (e.g., in the case of a low-latency chat application). The protocol is bootstrapped using a keyed challenge-response handshake, which looks sort of like HTTP and which is (quite remarkably) impossible to spoof by merely exploiting a header-splitting flaw in the destination site. Following a successful handshake, raw data may be exchanged bidirectionally within the resulting long-lived TCP connection, with each message enveloped inside a simple protocol frame. The mechanism is supported in WebKit and is probably coming soon to Firefox.

P2P networking

WebRTC[5] is a proposed set of APIs and network protocols designed to facilitate the discovery of and communication with other browsers without the need for a centralized server infrastructure. The primary use case for such a protocol is the implementation of IP telephony and video-conferencing features within web apps. No stable browser support is available yet.

Offline applications

Cache manifests[6] are a relatively simple way for a web server to instruct the browser that copies of certain documents should be stored indefinitely and reused whenever the client appears to have no network connectivity. In conjunction with client-side storage mechanisms such as *localStorage* (Chapter 9), this allows certain self-sufficient JavaScript applications to be used in offline mode. Offline operation is supported in Firefox, the WebKit browser, and Opera. As with *localStorage*, the persistent nature of this mechanism could exacerbate the long-term consequences of visiting an untrusted network.

Better cookies

Cake[7] is a now-expired proposal drafted by Adam Barth that aims to create a more lightweight and secure alternative to HTTP cookies: one origin-bound, browser-generated nonce for every destination site. A more current but incomplete proposal appears to flirt with normal but origin-based cookies as an alternative. Neither approach is available in any browser today.

Content-Level Features

The proposals outlined in this section aim to enable new classes of web applications to be built on top of HTML and JavaScript.

Client-side databases

Several APIs for creating and manipulating locally stored databases have been proposed over the years, including the notorious *WebSQL* API,[8] which would have brought the famously dangerous SQL syntax to client-side JavaScript. The WebSQL proposal was ditched in favor of a more sensible *IndexedDB* design,[9] which offers a clean API without serialized queries and has a security model comparable to that of *localStorage*—but not until WebSQL support had shipped in a couple of browsers. Meanwhile, the new API has shipped in Chrome and is expected to appear in Firefox.

Background processes

The *Worker* API,[10] available in Firefox, WebKit, and Opera, permits the creation of background JavaScript processes to perform computationally expensive tasks without having to worry about blocking the browser UI. Each worker runs in an isolated environment that lacks the usual *window* or *document* DOM and may communicate with its creator asynchronously through the *postMessage(…)* API. *Dedicated workers* are directly reachable only by their creator, while *shared workers* may be "attached" to several different sites at any given time. (*Persistent workers*, which would run independently of any sustained demand for their services, were proposed early on but then dropped.) The concept of worker threads raises some peripheral DoS concerns but otherwise poses no apparent security risks.

Geolocation discovery

The *navigator.geolocation.getCurrentPosition(…)* API[11] permits any website to request information about the physical location of the client device, subject to a user's (largely hijackable) consent. The computed geolocation data may be derived from GPS information on a system with a suitable hardware module, or it may be looked up based on the names of nearby wireless access points, cell towers, and so forth. The API is supported in all major browsers except for Internet Explorer.

Device orientation

A nonrestricted event-driven *DeviceOrientation* API[12] allows websites to read back the orientation of the device, based on accelerometer data. This API, which is probably geared toward mobile gaming, is available in Firefox, WebKit, and Opera on systems equipped with the appropriate hardware. Two researchers at the University of California, Davis have recently demonstrated a fatal flaw: On smartphones, minute movements of the device may be used to reliably reconstruct on-screen keyboard input, including passwords entered on unrelated websites.[13]

Page prerendering

This experimental feature in Chrome allows pages to be prefetched in anticipation of the user following a particular link, and it permits the entire HTML document to be prerendered in a hidden tab[14] and momentarily

revealed once the predicted navigation action takes place. The mechanism has some interesting browser security consequences if the prerendered page turns out to be malicious. The implementation in Chrome is careful to defer any disruptive actions until the tab is revealed, but mistakes will be very easy to make across all browser codebases.

Navigation timing
Several complementary APIs, currently available only in Chrome, permit certain types of navigation, including cross-domain page loads, to be very accurately benchmarked from client-side JavaScript.[15] This interface is designed to allow site owners to identify obvious performance bottlenecks, as experienced by a typical visitor. The API allows some privacy-related information to be collected by profiling the time needed to load certain third-party content, but because the same attack is possible in many other ways (for example with *onload* handlers on subresources), that probably does not matter much.

I/O Interfaces

The features listed below offer new input and output capabilities to web-based scripts.

UI notifications
Notification and *window.notifications*[16] APIs allow the creation of text-only or HTML-based, always-on-top pop-ups in the corner of the screen, allowing select web applications to gently notify users of important developments (such as a new mail message). User consent to receiving notifications is required on a per-site basis, limiting the risk of abuse. Nevertheless, care must be taken to properly communicate the origin of the tiny notification window and any dialogs or prompts it subsequently creates, an aspect that took some time to refine. The API is available only in WebKit today.

Full-screen mode
Several proposals have been circulated to allow JavaScript to maximize the current browser window and hide all the browser chrome. This functionality is essential to tasks such as viewing presentations or watching movies, but it is obviously very dangerous from the security standpoint: Once in control of the entire screen, any malicious page may draw a fake browser window with a fake address bar. So far, no specific implementation seems to be available for review. An early-stage proposal for mouse cursor locking is being discussed, too.

Media capture
A proposed suite of *navigator.device.capture* APIs[17] has been postulated for giving websites access to webcam and microphone data. Obvious security and privacy concerns arise around this mechanism, especially around the resilience of any associated security prompts with respect to race condition attacks. The API has no stable browser support today.

18

COMMON WEB VULNERABILITIES

Up until this point, we have paid little attention to the taxonomy of common web vulnerabilities. Gaining insight into the underlying mechanics of web applications is far more important than memorizing several thousand random and often unnecessary terms; nomenclature such as *improper restriction of operations within the bounds of a memory buffer* (Common Weakness Enumeration) or *insecure direct object references* (Open Web Application Security Project) finds no place in a reasonable conversation—and rightly so.

Nevertheless, the industry has come up with a handful of reasonably precise phrases that security researchers use every day. Having thoroughly discussed the inner workings of the browser, it seems useful to recap and highlight the terminology the average reader is likely to see.

Vulnerabilities Specific to Web Applications

The terms outlined in this section are unique to the technologies used on the Web and often have no immediate counterparts in the world of "traditional" application security.

Cross-site request forgery (XSRF, CSRF)
A vulnerability caused by the failure to verify that a particular state-changing HTTP request received by the server-side portion of the web application was initiated from the expected client-side origin. This flaw permits any third-party website loaded in the browser to perform actions on behalf of the victim.

⇨ *See Chapter 4 for a more detailed discussion of XSRF.*

Cross-site script inclusion (XSSI)
A flaw caused by the failure to secure sensitive JSON-like responses against being loaded on third-party sites via *<script src=...>*. User-specific information in the response may be leaked to attackers.

⇨ *See Chapter 6 for an overview of the problem (and potential fixes).*

Cross-site scripting (XSS)
Insufficient input validation or output escaping can allow an attacker to plant his own HTML markup or scripts on a vulnerable site. The injected scripts will have access to the entirety of the targeted web application and, in many cases, to HTTP cookies stored by the client.

The qualifier *reflected* refers to cases where the injected string is simply a result of incorrectly echoing back a portion of the request, whereas *stored* or *persistent* refers to a situation where the payload takes a more complex route. *DOM-based* may be used to denote that the vulnerability is triggered by the behavior of the client-side portion of the web app (i.e., JavaScript).

⇨ *See Chapter 4 for common XSS vectors in HTML documents.*

⇨ *See Chapter 6 for an overview of DOM-based XSS risks.*

⇨ *See Chapter 13 for XSS vectors associated with content sniffing.*

⇨ *See Chapter 9 for a discussion of the normal security model for JS code.*

Header injection (response splitting)
Insufficient escaping of newlines (or equivalent characters) in HTTP responses generated by the server-side portion of a web application. This behavior will typically lead to XSS, browser, or proxy cache poisoning and more.

⇨ *See Chapter 3 for a detailed discussion of the flaw.*

Mixed content
A catch-all name for loading non-HTTPS subresources on HTTPS pages. In the case of scripts and applets, this behavior makes the application trivially vulnerable to active attackers, particularly on open wireless

networks (at cafés, airports, and so on), and undoes almost all benefits of HTTPS. The consequences of mixed content bugs with stylesheets, fonts, images, or frames are usually also fairly serious but more constrained.

⇨ *See Chapters 4 and 8 for content-specific precautions on HTTPS sites.*

⇨ *See Chapter 11 for an overview of mixed-content handling rules.*

Open redirection
: A term used to refer to applications that perform HTTP- or script-based requests to user-supplied URLs without constraining the possible destinations in any meaningful way. Open redirection is not advisable and may be exploitable in some scenarios, but it is typically not particularly dangerous by itself.

⇨ *See Chapter 10 for cases where unconstrained redirection may lead to XSS.*

***Referer* leakage**
: Accidental disclosure of a sensitive URL by embedding an off-site subresource or providing an off-site link. Any security- or privacy-relevant data encoded in the URL of the parent document will be leaked in the *Referer* header, with the exception of the fragment identifier.

⇨ *See Chapter 3 for an overview of the* Referer *logic.*

Problems to Keep in Mind in Web Application Design

The problems outlined in this section are an unavoidable circumstance of doing business on the Internet and must be properly accounted for when designing or implementing new web apps.

Cache poisoning
: The possibility of long-term pollution of the browser cache (or any interim proxies) with a fabricated, malicious version of the targeted web application. Encrypted web applications may be targeted due to response-splitting vulnerabilities. For nonencrypted traffic, active network attackers may be able to modify the responses received by the requestor, too.

⇨ *See Chapter 3 for an overview of HTTP-caching behaviors.*

Clickjacking
: The possibility of framing or otherwise decorating or obscuring a portion of another web application so that the victim, when interacting with the attacker's site, is not aware that individual clicks or keystrokes are delivered to the other site, resulting in undesirable actions being taken on behalf of the user.

⇨ *See Chapter 11 for a discussion of clickjacking and related UI issues.*

Content and character set sniffing

Describes the possibility that the browser will ignore any authoritative content type or character set information provided by the server and interpret the returned document incorrectly.

⇨ *See Chapter 13 for a discussion of content-sniffing logic.*

⇨ *See Chapters 4 and 8 for scenarios where* Content-Type *data is ignored.*

Cookie forcing (or cookie injection)

The possibility of blindly injecting HTTP cookies into the context of an otherwise impenetrable web application due to issues in how the mechanism is designed and implemented in modern browsers. Cookie injection is of particular concern to HTTPS applications. (*Cookie stuffing* is a less common term referring specifically to maliciously deleting cookies belonging to another application by overflowing the cookie jar.)

⇨ *See Chapter 9 for more information on cookie scoping.*

⇨ *See Chapter 3 for a general discussion of the operation of HTTP cookies.*

Denial-of-service (DoS) attacks

A broad term denoting any opportunities for the attacker to bring down a browser or server or otherwise make the use of a particular targeted application significantly more difficult.

⇨ *See Chapter 14 for an overview of DoS considerations with JavaScript.*

Framebusting

The possibility of a framed page navigating the top-level document to a new URL without having to satisfy same-origin checks. The behavior may be exploited for phishing attacks or simply for petty mischief.

⇨ *See Chapter 11 for this and other frame navigation quirks.*

HTTP downgrade

The ability for active attackers to prevent the user from reaching an HTTPS version of a particular site or to downgrade an existing HTTPS session to HTTP.

⇨ *See Chapter 3 for an overview of HTTPS.*

⇨ *See Chapter 16 for Strict Transport Security, a proposed solution to the problem.*

Network fenceposts

The prospect of websites on the Internet leveraging the browser to interact with destinations not directly accessible to the attacker, for example, with the systems on a victim's internal network. Such attacks can be performed blindly, or (with the help of attacks such as DNS rebinding) the attacker may be able to see responses to all requests.

⇨ *See Chapter 12 for an overview of non-SOP boundaries in a browser.*

⇨ *See Chapter 15 for Internet Explorer zone model, a potential approach to this risk.*

NOTE ***Beware non-buzzword bugs!*** *Not all vulnerabilities have catchy names. Web developers should be wary of many other implementation and design issues that are outside the scope of this book but that can nevertheless bite hard. Examples include weak pseudo-random number generators (especially for session management purposes); insufficient authentication and authorization checks (in particular, overly trusting the browser-originating data); incorrect uses of cryptography (inventing one's own algorithms is usually a no-no); and so on. For a remarkably detailed discussion of these and many other failure patterns, see* The Art of Software Security Assessment *by Dowd, McDonald, and Schuh (Addison-Wesley, 2006).*

Common Problems Unique to Server-Side Code

The following issues are commonly encountered in the server-hosted portion of any web application and, by virtue of being tied to specific programming languages or software components, are unlikely to occur on the client side.

Buffer overflow
: A condition where a program allows more information to be stored in a particular memory region than there is space to accommodate the incoming data, leading to the unexpected overwrite of other vital data structures. Buffer overflows happen chiefly in low-level programming languages, such as C or C++, and in these languages, they can be frequently leveraged to execute attacker-supplied code.

Command injection (SQL, shell, PHP, and so on)
: A problem where, due to insufficient input filtering or output escaping, attacker-controlled strings may be unintentionally processed as statements in an interpreted language used by the application. (In a distant sense, this is similar to XSS.) The consequences depend on the capabilities of the language, but in most cases, code execution is the eventual outcome.

Directory traversal
: A problem where, due to insufficient input filtering (most commonly, the failure to properly recognize and handle "../" segments in filenames), an application can be tricked into reading or writing files at arbitrary locations on the disk. Any consequences depend on additional constraints, but unconstrained file-writing bugs are usually easily exploitable to run attacker-supplied code.

File inclusion
: If used without a qualifier or prefixed with *local (LFI)*, the term is largely synonymous with read-related directory traversal. *Remote file inclusion (RFI)*, on the other hand, is an alternative way to exploit file-inclusion vulnerabilities by specifying a URL rather than a valid file path. In some scripting languages, a single, common API opens local files and fetches remote URLs. In these cases, the ability to retrieve the file from an attacker-controlled server may offer substantial benefits, depending on how the data is subsequently processed.

Format-string vulnerability

A handful of commonly used library functions accept templates ("format strings"), followed by a set of parameters that the function is expected to insert into the template at predefined locations. Such an approach is particularly common in C (*printf(...)*, *syslog(...)*, and so on), but it is not limited to that language. Format-string vulnerabilities are caused by unintentionally permitting attackers to supply the template to one of these functions. Depending on the capabilities of the template system and the specifics of the language, this error may lead to anything from minor data leaks to code execution.

Integer overflow

A vulnerability specific to languages with limited or no range checking. The flaw is caused by the developer failing to detect that an integer exceeded the maximum possible value and rolled back to zero, to a very large negative integer, or to some other hardware-specific and unexpected result. Depending on how the value is used, this may put the program in an inconsistent state or, worse, lead to the reading or writing of data at an incorrect memory location (which, in turn, may lend itself to code execution). Integer *underflow* is the opposite effect: crossing the minimum permissible value and rolling over to a very large positive integer.

Pointer management vulnerabilities

In languages that encourage or require the use of raw memory pointers (chiefly C and C++), it is possible to use pointers that are either uninitialized or no longer valid ("dangling"), leading to vulnerabilities such as *use after free*, *double free*, and many more. These vulnerabilities will corrupt the internal state of the program and usually allow an attacker to execute attacker-supplied code.

EPILOGUE

Well, who would have thought. This concludes *The Tangled Web*! I hope you've enjoyed reading this book as much as I've enjoyed exploring the world of browser security over the last decade or so. I also hope that what you've discovered on these pages will guide you in your future journeys, wherever they may be.

As for what to make of it all: To me, the stark contrast between the amazing robustness of the modern Web and the inexplicable unsoundness of its foundations was difficult to reconcile at first. In retrospect, I think it offers an important insight into our own, unreasonable attitude about securing the online world.

I am haunted by the uncomfortable observation that in real life, modern societies are built on remarkably shaky ground. Every day, each of us depends on the sanity, moral standards, and restraint of thousands of random strangers—from cab drivers, to food vendors, to elevator repair techs. The rules of the game are weakly enforced through a series of deterrence

mechanisms, but if crime statistics are to be believed, their efficacy is remarkably low. The problem isn't just that most petty criminals think they can get away with their misdeeds but that they are usually right.

In this sense, our world is little more than an incredibly elaborate honor system that most of us voluntarily agree to participate in. And that's probably okay: Compliance with self-imposed norms has proven to be a smart evolutionary move, and it is a part of who we are today. A degree of trust is simply essential to advancing our civilization at a reasonable pace. Too, paradoxically, despite short-term weaknesses, accelerated progress makes us all a lot stronger and more adaptable in the long run.

It is difficult to understand, then, why we treat our online existence in such a dramatically different way. For example, why is it that we get upset at developers who use cryptography incorrectly, but we don't mind that the locks on our doors can be opened with a safety pin? Why do we scorn web developers who can't get input validation right, but we don't test our breakfast for laxatives or LSD?

The only explanation I can see is that humankind has had thousands of years to work out the rules of social engagement in the physical realm. During that time, entire societies have collapsed, new ones have emerged, and an increasingly complex system of behavioral norms, optimized for the preservation of communities, has emerged in the process. Unfortunately for us, we have difficulty transposing these rules into the online ecosystem, and this world is so young, it hasn't had the chance to develop its own, separate code of conduct yet.

The phenomenon is easy to see: While your neighbor will not try to sneak into your house, he may have no qualms about using your wireless network because doing so feels much less like a crime. He may oppose theft, but he may be ambivalent about unlawfully duplicating digital content. Or he may frown upon crude graffiti in the neighborhood but chuckle at the sight of a defaced website. The parallels are there but just aren't good enough.

What if our pursuit of perfection in the world of information security stems from nothing but a fundamental misunderstanding of how human communities can emerge and flourish? The experts of my kind preach a model of networked existence based on complete distrust, but perhaps wrongly so: As the complexity of our online interactions approaches that of real life, the odds of designing perfectly secure software are rapidly diminishing. Meanwhile, the extreme paranoia begins to take a heavy toll on how quickly we can progress.

Perhaps we are peddling a recipe for a failure. What if our insistence on absolute security only takes us closer to the fate of so many other early civilizations, which collapsed under the weight of their flaws and ultimately vanished? I find this perverse thought difficult to dismiss. Fortunately, we know that from the rubble, new, more enlightened societies will certainly emerge one day. Their ultimate nature is anyone's guess.

NOTES

Chapter 1

1. D.E. Bell and L.J. La Padula, *Secure Computer System: Unified Exposition and Multics Interpretation* (ESD-TR-75-306), Bedford, MA: MITRE Corporation for US Air Force (1976), *http://csrc.nist.gov/publications/history/bell76.pdf.*
2. C.E. Landwehr, C.L. Heitmeyer, and J.D. McLean, "A Security Model for Military Message Systems: Retrospective," paper presented at the 17th Annual Computer Security Applications Conference, New Orleans, LA (2001), *http://www.acsa-admin.org/2001/papers/141.pdf.*
3. V. Bush, "As We May Think," *Atlantic Monthly* (July 1945), *http://www.theatlantic.com/doc/194507/bush/.*
4. R. Dhamija, J.D. Tygar, and M. Hearst, "Why Phishing Works," paper presented at the Conference on Human Factors in Computing Systems, Montreal, Canada (2006), *http://people.seas.harvard.edu/~rachna/papers/why_phishing_works.pdf.*
5. C. Jackson, D.R. Simon, D.S. Tan, and A. Barth, "An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks," paper presented at Usable Security, Lowlands, Trinidad and Tobago (2007), *http://usablesecurity.org/papers/jackson.pdf.*
6. C. Jackson and A. Barth, "Beware of Finer-Grained Origins," paper presented at Web 2.0 Security and Privacy, Oakland, CA (2008), *http://seclab.stanford.edu/websec/origins/fgo.pdf*; C. Jackson, and A. Barth, "Beware of Coarser-Grained Origins," paper presented at Web 2.0 Security and Privacy, Oakland, CA (2008), *http://seclab.stanford.edu/websec/origins/scheme/.*

7. "Security Exploit Uses Internet Explorer to Attack Mozilla Firefox," MozillaZine (July 11, 2007), *http://www.mozillazine.org/talkback.html?article=22198*.

Page 19

1. Net Applications website, *http://marketshare.hitslink.com/browser-market-share.aspx?qprid=0, http://marketshare.hitslink.com/browser-market-share.aspx?qprid=2* (accessed June 13, 2011).

Chapter 2

1. T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," IETF Request for Comments 3986 (2005), *http://www.ietf.org/rfc/rfc3986.txt*.
2. T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," IETF Request for Comments 1738 (1994), *http://www.ietf.org/rfc/rfc1738.txt*.
3. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol—HTTP/1.1," IETF Request for Comments 2616 (1999), *http://www.ietf.org/rfc/rfc2616.txt*.
4. "Uniform Resource Identifer (URI) Schemes per RFC4395," Internet Assigned Numbers Authority (June 6, 2011), *http://www.iana.org/assignments/uri-schemes.html*.
5. P. Mockapetris, "Domain Names—Implementation and Specification," IETF Request for Comments 1035 (1987), *http://www.ietf.org/rfc/rfc1035.txt*.
6. T. Berners-Lee, "Universal Resource Identifiers in WWW," IETF Request for Comments 1630 (1994), *http://www.w3.org/Addressing/rfc1630.txt*.
7. P. Hoffman, L. Masinter, and J. Zawinski, "The mailto URL Scheme," IETF Request for Comments 2368 (1998), *http://www.ietf.org/rfc/rfc2368.txt*.
8. "HTML 4.01 Specification: Forms," World Wide Web Consortium (1999), *http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1*.
9. P. Faltstrom, P. Hoffman, and A. Costello, "Internationalizing Domain Names in Applications (IDNA)," IETF Request for Comments 3490 (2003), *http://www.ietf.org/rfc/rfc3490.txt*.
10. A. Costello, "Punycode: A Bootstring Encoding of Unicode for Internationalized Domain Names in Applications (IDNA)," IETF Request for Comments 3492 (2003), *http://www.ietf.org/rfc/rfc3492.txt*.
11. E. Gabrilovich and A. Gontmakher, "The Homograph Attack," Communications of the ACM (2002), *http://www.cs.technion.ac.il/~gabr/papers/homograph_full.pdf*.

12. E. Rescorla, "HTTP Over TLS," IETF Request for Comments 2818 (2000), *http://www.ietf.org/rfc/rfc2818.txt*.
13. J. Postel and J. Reynolds, "File Transfer Protocol (FTP)," IETF Request for Comments 959 (1985), *http://www.ietf.org/rfc/rfc959.txt*.
14. F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti, "The Internet Gopher Protocol," IETF Request for Comments 1436 (1993), *http://www.ietf.org/rfc/rfc1436.txt*.
15. E. Rescorla and A. Schiffman, "The Secure HyperText Transfer Protocol," IETF Request for Comments 2660 (1999), *http://www.ietf.org/rfc/rfc2660.txt*.
16. L. Masinter, "The 'data' URL Scheme," IETF Request for Comments 2397 (1998), *http://www.ietf.org/rfc/rfc2397.txt*.
17. "What Are rss: and feed: Links?" *http://www.brindys.com/winrss/feedformat.html*.
18. M. Zalewski, "A Note on an MHTML Vulnerability," *Lcamtuf's blog* (March 11, 2011), *http://lcamtuf.blogspot.com/2011/03/note-on-mhtml-vulnerability.html*.

Chapter 3

1. T. Berners-Lee, "The Original HTTP as defined in 1991." World Wide Web Consortium archives (1991), *http://www.w3.org/Protocols/HTTP/AsImplemented.html*.
2. T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol—HTTP/1.0," IETF Request for Comments 1945 (1996), *http://www.ietf.org/rfc/rfc1945.txt*.
3. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol—HTTP/1.1," IETF Request for Comments 2616 (1999), *http://www.ietf.org/rfc/rfc2616.txt*.
4. HTTPbis Working Group, "Httpbis Status Pages," *http://tools.ietf.org/wg/httpbis/*.
5. A. Luotonen, "Tunneling TCP-Based Protocols Through Web Proxy Servers," IETF draft (1998), *http://tools.ietf.org/id/draft-luotonen-web-proxy-tunneling-01.txt*.
6. S. Chen, Z. Mao, Y.M. Wang, and M. Zhang, "Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments," Microsoft Research (2009), *http://research.microsoft.com/pubs/79323/pbp-final-with-update.pdf*.
7. "Mozilla Cross-Reference mozilla1.8.0," Mozilla code repository, *http://mxr.mozilla.org/mozilla1.8.0/source/nsprpub/pr/src/misc/prtime.c#1045*.
8. K. Moore, "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text," IETF Request For Comments 2047 (1996), *http://www.ietf.org/rfc/rfc2047.txt*.

9. N. Freed and K. Moore, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations," IETF Request for Comments 2231 (1997), *http://www.ietf.org/rfc/rfc2231.txt*.
10. Mozilla Bug Tracking System, Mozilla bug #418394, *https://bugzilla.mozilla.org/show_bug.cgi?id=418394*.
11. T. Berners-Lee, "Basic HTTP as defined in 1992: Methods," World Wide Web Consortium archives (1992), *http://www.w3.org/Protocols/HTTP/Methods.html*.
12. L. Dusseault, "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)," IETF Request for Comments 4918 (2007), *http://www.ietf.org/rfc/rfc4918.txt*.
13. See note 12 above.
14. M. Pool, "Meantime: Non-Consensual HTTP User Tracking Using Caches" (2000), *http://sourcefrog.net/projects/meantime/*.
15. L. Montulli, "Persistent Client State HTTP Cookies" (1994), *http://curl.haxx.se/rfc/cookie_spec.html*.
16. D. Kristol and L. Montulli, "HTTP State Management Mechanism," IETF Request for Comments 2109 (1997), *http://www.ietf.org/rfc/rfc2109.txt*.
17. D. Kristol and L. Montulli, "HTTP State Management Mechanism," IETF Request for Comments 2965 (2000), *http://tools.ietf.org/rfc/rfc2965.txt*.
18. A. Barth, "HTTP State Management Mechanism," IETF Request for Comments 6265 (2011), *http://www.ietf.org/rfc/rfc6265.txt*.
19. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," IETF Request for Comments 2617 (1999), *http://www.ietf.org/rfc/rfc2617.txt*.
20. R. Tschalär, "NTLM Authentication Scheme for HTTP" (2003), *http://www.innovation.ch/personal/ronald/ntlm.html*.
21. E. Rescorla, "HTTP Over TLS," IETF Request for Comments 2818 (2000), *http://www.ietf.org/rfc/rfc2818.txt*.
22. P. Hallam-Baker, "The Recent RA Compromise," *Comodo IT Security* (blog) (March 23, 2011), *http://blogs.comodo.com/it-security/data-security/the-recent-ra-compromise/*.
23. S. Chen, R. Wang, X. F. Wang, and K. Zhang, "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," Microsoft Research (2010), *http://research.microsoft.com/pubs/119060/WebAppSideChannel-final.pdf*.
24. C. Evans, "Open Redirectors: Some Sanity," *Security: Hacking Everything* (blog) (June 25, 2010), *http://scarybeastsecurity.blogspot.com/2010_06_01_archive.html*.

Chapter 4

1. T. Berners-Lee, "HTML Tags," World Wide Web Consortium archives (1991), *http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html*.
2. T. Berners-Lee and D. Connolly, "Hypertext Markup Language—2.0," IETF Request for Comments 1866 (1995), *http://www.ietf.org/rfc/rfc1866.txt*.
3. D. Raggett, "HTML 3.2 Reference Specification," World Wide Web Consortium (1997), *http://www.w3.org/TR/REC-html32*.
4. D. Raggett, A. Le Hors, and I. Jacobs, "HTML 4.01 Specification," World Wide Web Consortium (1999), *http://www.w3.org/TR/html401/*.
5. I. Hickson, "HTML5," World Wide Web Consortium draft, revision 1.5019 (2011), *http://dev.w3.org/html5/spec/Overview.html*.
6. G. Coldwind, "Too general charset = detection in meta," Mozilla bug 640529 (2011), *https://bugzilla.mozilla.org/show_bug.cgi?id=640529*.

Chapter 5

1. H. Wium Lie and B. Bos, "Cascading Style Sheets, Level 1," World WideWeb Consortium, (1996), *http://www.w3.org/TR/CSS1/*.
2. T. Çelik, E.J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams, "Selectors Level 3: Selectors," World Wide Web Consortium (2009), *http://www.w3.org/TR/css3-selectors/#selectors*.
3. I. Hickson, "XML Binding Language (XBL) 2.0," World Wide Web Consortium (2007), *http://www.w3.org/TR/xbl/*.
4. G. Heyes, D. Lindsay, and E.V. Nava, "The Sexy Assassin: Tactical Exploitation Using CSS" (2009), *http://www.scribd.com/doc/54664700/Tactical-Xploit-Css*.

Chapter 6

1. Netscape Communications Corporation, "Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet" (press release) (December 4, 1995), *http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html*.
2. ECMA International, "ECMA-262: ECMAScript Language Specification," 3rd ed. (1999), *http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf*.
3. ECMA International, "ECMA-262: ECMAScript Language Specification," 5th ed. (2009), *http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf*.

4. D. Crockford, "The Application/JSON Media Type for JavaScript Object Notation (JSON)," IETF Request for Comments 4627 (2006), *http://www.ietf.org/rfc/rfc4627.txt*.
5. J. Schneider, R. Yu, and J. Dyer, eds., "Standard ECMA-357: ECMAScript for XML (E4X) Specification," 2nd ed., ECMA International (2005), *http://www.ecma-international.org/publications/standards/Ecma-357.htm*.
6. P. Le Hégaret, R. Whitmer, and L. Wood, "Document Object Model (DOM)," World Wide Web Consortium (2005), *http://www.w3.org/DOM/*.
7. E. Vela Nava, "Bug 38922—innerHTML decompilation issues in textarea" (WebKit bug-tracking system post) (2010), *https://bugs.webkit.org/show_bug.cgi?id=38922*.
8. "Windows Scripting 5.8: MsgBox Function," Microsoft Developer Network Platforms (2009), *http://msdn.microsoft.com/en-us/library/sfw6660x%28v=vs.85%29.aspx*.
9. D. Crockford, "JSON in JavaScript," *GitHub Social Coding* (blog) (March 5, 2011), *https://github.com/douglascrockford/JSON-js/blob/master/json2.js*.

Chapter 7

1. J. Ferraiolo, F. Jun, and D. Jackson, "Scalable Vector Graphics (SVG) 1.1 Specification," World Wide Web Consortium (2003), *http://www.w3.org/TR/2003/REC-SVG11-20030114/*.
2. D. Carlisle, P. Ion, and R. Miner, "Mathematical Markup Language (MathML) Version 3.0," World Wide Web Consortium WC3 Recommendation 21 (2010), *http://www.w3.org/TR/MathML3/*.
3. A. Mechelynck, "XUL," Mozilla Developer Network (2011), *https://developer.mozilla.org/en/xul*.
4. Wireless Application Protocol Forum, "Wireless Application Protocol: Wireless Markup Language Specification version 30" (1998), *http://www.wapforum.org/what/technical/wml-30-apr-98.pdf*.
5. RSS Advisory Board, "RSS 2.0 Specification version 2.0.11" (2009), *http://www.rssboard.org/rss-specification*.
6. M. Nottingham and R. Sayre, eds., "The Atom Syndication Format," IETF Request for Comments 4287 (2005), *http://www.ietf.org/rfc/rfc4287.txt*.

Chapter 8

1. E. Mills, "Security Labs Report: January–June 2010 Recap," M86 Security (2010), *http://www.m86security.com/documents/pdfs/security_labs/m86_security_labs_report_1H2010.pdf*.
2. B. Rios, "Sun Fixes GIFARs" (December 17, 2008), *http://xs-sniper.com/blog/2008/12/17/sun-fixes-gifars/*.

3. A.K. Sood, "PDF Silent HTTP Form Repurposing Attacks," SecNiche Security Labs (2009), *http://secniche.org/papers/SNS_09_03_PDF_Silent_Form_Re_Purp_Attack.pdf*.
4. P.D. Petkov, "Universal PDF XSS Afterparty" (January 4, 2007), *http://www.gnucitizen.org/blog/universal-pdf-xss-after-party/*.
5. S. Jobs, "Thoughts on Flash" (2010), *http://www.apple.com/hotnews/thoughts-on-flash/*.
6. "Adobe Shockwave Player," Adobe Systems Incorporated, *http://www.adobe.com/products/shockwaveplayer/*.
7. "ActionScript 3.0 Reference for the Adobe Flash Platform," Adobe Systems Incorporated, *http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html*.
8. "Content Played Back in Flash Player Reaches 99% of Internet Viewers," Adobe Systems Incorporated (March 2011), *http://www.adobe.com/products/player_census/flashplayer/*.
9. "Web Browser Plugin Market Share," StatOwl (May 2011), *http://www.statowl.com/plugin_overview.php*.
10. "ActionScript 3.0 Reference for the Adobe Flash Platform: ExternalInterface," Adobe Systems Incorporated, *http://livedocs.adobe.com/flex/3/langref/flash/external/ExternalInterface.html#includeExamplesSummary*.
11. "XAML Overview (WPF)," Microsoft Corporation, *http://msdn.microsoft.com/en-us/library/ms752059.aspx*.
12. "Rich Internet Application Statistics" (July 2011), *http://www.riastats.com/*. See also StatOwl (Chapter 8, note 9).
13. "Secunia Half Year Report 2010," Secunia (2010), *http://secunia.com/gfx/pdf/Secunia_Half_Year_Report_2010.pdf*.
14. "WPF XAML Browser Applications Overview," Microsoft Corporation, *http://msdn.microsoft.com/en-us/library/aa970060.aspx*.
15. "Akamai Download Manager Help," Microsoft Corporation, *https://msdn.microsoft.com/en-us/subscriptions/manage/bb153537.aspx*.

Chapter 9

1. A. Klein, "IE + Some Popular Forward Proxy Servers = XSS, Defacement (Browser Cache Poisoning)" (May 22, 2006), *http://seclists.org/webappsec/2006/q2/352*; M. Zalewski, "Web 2.0 Backdoors Made Easy with MSIE & XMLHttpRequest" (February 3, 2007), *http://seclists.org/fulldisclosure/2007/Feb/81*.
2. A. van Kesteren, ed., "Cross-Origin Resource Sharing," working draft, World Wide Web Consortium (July 27, 2010), *http://www.w3.org/TR/cors/*.
3. I. Hickson, "Web Storage," editor's draft, World Wide Web Consortium (July 28, 2011), *http://dev.w3.org/html5/webstorage/*.

4. J. Stenback, "Make sessionStorage Use Principals Instead of String Domains," Mozilla bug #495337 (May 28, 2009), *https://bugzilla.mozilla.org/show_bug.cgi?id=495337*.
5. T. Ormandy, "Common DNS Misconfiguration Can Lead to 'Same Site' Scripting" (January 18, 2008), *http://seclists.org/bugtraq/2008/Jan/270*.
6. R. Singel, "ISPs' Error Page Ads Let Hackers Hijack Entire Web, Researcher Discloses," *Wired* (April 19, 2008), *http://www.wired.com/threatlevel/2008/04/isps-error-page/*.
7. "APSB10-14 Security Update Available for Adobe Flash Player," Adobe Systems Incorporated (June 10, 2010), *http://www.adobe.com/support/security/bulletins/apsb10-14.html*.
8. "Understanding Flash Player 9 April 2008: Security Update Compatibility," Adobe Systems Incorporated (April 8, 2008), *http://www.adobe.com/devnet/flashplayer/articles/flash_player9_security_update.html*.
9. "ActionScript® 3.0 Reference for the Adobe® Flash® Platform: URLRequestHeader," Adobe Systems Incorporated, *http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/URLRequestHeader.html*.
10. "ActionScript® 3.0 Reference for the Adobe® Flash® Platform: Security," Adobe Systems Incorporated, *http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/system/Security.html*.
11. "Adobe Cross Domain Policy File Specification," version 2.0, Adobe Systems Incorporated (August 2, 2010), *http://learn.adobe.com/wiki/download/attachments/64389123/CrossDomain_PolicyFile_Specification.pdf?version=1*.
12. M. Zalewski, "[RAZOR] Linux Kernel IP Masquerading Vulnerability" (July 30, 2001), *http://seclists.org/bugtraq/2001/Jul/733*.
13. "Silverlight: WebHeaderCollection Class," Microsoft, *http://msdn.microsoft.com/en-us/library/system.net.webheadercollection%28v=VS.95%29.aspx*.
14. "Class HttpURLConnection," Sun Microsystems/Oracle, *http://download.oracle.com/javase/1.4.2/docs/api/java/net/HttpURLConnection.html*.
15. "Class URLConnection," Sun Microsystems/Oracle, *http://download.oracle.com/javase/1.4.2/docs/api/java/net/URLConnection.html*.
16. "Class Socket," Sun Microsystems/Oracle, *http://download.oracle.com/javase/1.4.2/docs/api/java/net/Socket.html*.
17. "Java-to-Javascript Communication," Sun Microsystems/Oracle, *http://download.oracle.com/javase/1.4.2/docs/guide/plugin/developer_guide/java_js.html*.
18. "Java-to-Javascript Communication: Common DOM API," Sun Microsystems/Oracle, *http://download.oracle.com/javase/1.4.2/docs/guide/plugin/developer_guide/java_js.html#common_dom*.
19. B. "Snowhare" Franz, "Triple Dot Cookies" (1998), *http://snowhare.com/utilities/triple_dot/*.
20. "Adobe ActionScript 3.0: Security Sandboxes," Adobe Systems Incorporated, *http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7e3f.html*.

Chapter 10

1. L. Masinter, "The 'data' URL scheme," IETF Request for Comments 2397 (1998), *http://www.ietf.org/rfc/rfc2397.txt*.
2. M. Zalewski, "about:neterror, certerror permit URL spoofing by being same-origin with about:blank," Mozilla bug #602780 (CVE-2010-3774) (2010), *https://bugzilla.mozilla.org/show_bug.cgi?id=602780*.

Chapter 11

1. G. Guninski, "Frame spoofing using loading two frames," Mozilla bug #13871 (1999), *https://bugzilla.mozilla.org/show_bug.cgi?id=13871*.
2. R. Zilberman, "Frame spoofing is possible within a short time frame while the window is loading," Mozilla bug #381300 (CVE-2007-3089) (2008), *https://bugzilla.mozilla.org/show_bug.cgi?id=381300*.
3. A. Barth, C. Jackson, and J.C. Mitchell, "Securing Frame Communication in Browsers," *Communications of the ACM 52*, no. 6 (2009): 83-91, *http://www.adambarth.com/papers/2009/barth-jackson-mitchell-cacm.pdf*.
4. R. Hansen and J. Grossman, "Clickjacking" (2008), *http://www.sectheory.com/clickjacking.htm*.
5. M. Zalewski, "Dealing with UI redress vulnerabilities inherent to the current web" (post to whatwg.org list) (September 25, 2008), *http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2008-September/thread.html#16292*.
6. E. Lawrence, "IE8 Security Part VII: ClickJacking Defenses," *IEBlog* (January 27, 2009), *http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx*.
7. SHODAN, "HTTP Header Survey" (March 14, 2011), *http://www.shodanhq.com/research/infodisc/report*.
8. P. Stone, "Next Generation Clickjacking," Blackhat Europe (2010), *http://blog.c22.cc/2010/04/14/blackhat-europe-next-generation-clickjacking-3/*.
9. M. Zalewski, "The curse of inverse strokejacking," *Icamtuf's blog* (June 8, 2010), *http://lcamtuf.blogspot.com/2010/06/curse-of-inverse-strokejacking.html*.
10. C. Evans, "Generic cross-browser cross-domain theft," *Security* (blog) (December 28, 2009) *http://scarybeastsecurity.blogspot.com/2009/12/generic-cross-browser-cross-domain.html*.
11. C. Evans, "IE8 CSS-based forced tweeting," *Security* (blog) (September 29, 2010), *http://scarybeastsecurity.blogspot.com/2010/09/ie8-css-based-forced-tweeting.html*.
12. I. Hickson, "HTML: 4.8.11 The canvas element," WHATWG (2011), *http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html*.
13. E.W. Felten and M.A. Schneider, "Timing Attacks on Web Privacy," *Proceedings of the 7th ACM Conference on Computer and Communications Security* (2000), *http://sip.cs.princeton.edu/pub/webtiming.pdf*.

14. C. Evans, "Cross-domain search timing," *Security* (blog) (December 11, 2009), *http://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html*.
15. C. Wilson, P. Le Hégaret, and V. Apparao, "Document Object Model CSS: 2.2.1 Override and computed style sheet," World Wide Web Consortium (2000), *http://www.w3.org/TR/DOM-Level-2-Style/css.html#CSS-OverrideAndComputed*.
16. "currentStyle Object," Microsoft Corporation MSDN Library, *http://msdn.microsoft.com/en-us/library/ms535231%28v=vs.85%29.aspx*.
17. A. Clover, "CSS visited pages disclosure" (February 20, 2002), *http://seclists.org/bugtraq/2002/Feb/271*.
18. Z. Weinberg, E.Y. Chen, P.R. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer" (2011), *http://websec.sv.cmu.edu/visited/visited.pdf*.

Chapter 12

1. J. Schwartz, "Giving Web a Memory Cost Its Users Privacy," *New York Times* (September 4, 2001), *http://www.nytimes.com/2001/09/04/technology/04COOK.html*.
2. N. Wingfield, "Microsoft Quashed Effort to Boost Online Privacy," *Wall Street Journal* (August 2, 2010), *http://online.wsj.com/article/SB10001424052748703467304575383530439838568.html*.
3. E. Felten, "If You're Going to Track Me, Please Use Cookies," *Freedom to Tinker* (blog) (July 7, 2009), *http://www.freedom-to-tinker.com/blog/felten/if-youre-going-track-me-please-use-cookies*.
4. J. Mayer, A. Narayanan, and S. Stamm, "Do Not Track: A Universal Third-Party Web Tracking Opt Out," IETF Request for Comments (2011), *http://datatracker.ietf.org/doc/draft-mayer-do-not-track/?include_text=1*.
5. L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle, "The Platform for Privacy Preferences 1.0 (P3P1.0) Specification," World Wide Web Consortium (2002), *http://www.w3.org/TR/P3P/*.

Chapter 13

1. N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," IETF Request for Comments 2046 (1996), *http://www.ietf.org/rfc/rfc2046.txt*.
2. V. Gupta, "IE Content-Type Logic," *IEBlog* (February 1, 2005), *http://blogs.msdn.com/b/ie/archive/2005/02/01/364581.aspx*.
3. SHODAN, "HTTP Header Survey" (2011), *http://www.shodanhq.com/research/infodisc/download_latest*.

4. R. Troost, S. Dorner, and K. Moore, "Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field," IETF Request for Comments 2183 (1997), *http://www.ietf.org/rfc/rfc2183.txt*.
5. G. Heyes, "Inline UTF-7 E4X Javascript Hijacking," *The Spanner* (blog) (February 24, 2009), *http://www.thespanner.co.uk/2009/02/24/inline-utf-7-e4x-javascript-hijacking/*.

Chapter 14

1. M. Zalewski, "URL Spoofing Is Likely Possible Through Address Bar Eliding" (2010), *https://bugzilla.mozilla.org/show_bug.cgi?id=581313*.
2. R. J. Kosinski, "A Literature Review on Reaction Time," Clemson University (2010), *http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm#Type%20of%20Stimulus*.
3. M. Zalewski, "Bug 376473: File Action Dialog Controls Vulnerable to Refocus Race" (2007), *https://bugzilla.mozilla.org/show_bug.cgi?id=376473*.
4. M. Zalewski, "Geolocation Spoofing and Other UI Woes," *Bugtraq* (mailing list) (August 17, 2010), *http://seclists.org/bugtraq/2010/Aug/201*.
5. D. Simons and C. Chabris, "Selective Attention Test" (1999), *http://www.youtube.com/watch?v=vJG698U2Mvo&feature=player_embedded*.
6. D.J. Simmons and C.F. Chabris, "Gorillas in our midst: Sustained inattentional blindness for dynamic events," *Perception*, 28, 1059–1074 (1999), *http://www.cnbc.cmu.edu/~behrmann/dlpapers/Simons_Chabris.pdf*.

Chapter 15

1. *http://www.xssed.com/search?key=addons.mozilla.org*
2. *http://openid.net/*
3. "Internet Explorer: Security Zones," Microsoft, *http://technet.microsoft.com/en-us/library/dd361896.aspx*.
4. "Internet Explorer Binary Behaviors Security Setting," Microsoft, *http://technet.microsoft.com/en-us/library/cc776248(WS.10).aspx*.
5. Charles Schwab, "Technical Support," *http://www.visualwebcaster.com/charles_schwab/support/* (accessed September 9, 2011).
6. Internal Revenue Service, "Streaming Media System Requirements & Troubleshooting Assistance," *http://www.irsvideos.gov/sbv_1099webinar/player/IRS_Webinar_Technical_Support.pdf* (accessed September 9, 2011).
7. ".NET Framework 3.0: Mark of the Web," Microsoft, *http://msdn.microsoft.com/en-us/library/ms537628%28VS.85%29.aspx*.
8. "Persistent Zone Identifier Object," Microsoft, *http://msdn.microsoft.com/en-us/library/ms537029%28VS.85%29.aspx*.

Chapter 16

1. A. van Kesteren, "Cross-Origin Resource Sharing," (working draft) World Wide Web Consortium (July 27, 2010), *http://www.w3.org/TR/cors/*.
2. S. Dutta, "Updates for AJAX in IE8 Beta 2," *IEBlog* (2008), *http://blogs.msdn.com/b/ie/archive/2008/10/06/updates-for-ajax-in-ie8-beta-2.aspx*.
3. ".NET Framework 3.0: XDomainRequest Object," Microsoft Developer Network, *http://msdn.microsoft.com/en-us/library/cc288060%28v=vs.85%29.aspx*.
4. T. Close and M. Miller, "Uniform Messaging Policy, Level One," (working draft) World Wide Web Consortium (January 26, 2010), *http://www.w3.org/TR/UMP/*.
5. A. Barth, C. Jackson, and J.C. Mitchell, "Robust Defenses for Cross-Site Request Forgery," ACM Conference on Computer and Communications Security (2008), *http://seclab.stanford.edu/websec/csrf/csrf.pdf*.
6. B. Sterne, "Origin Header Proposal," *http://people.mozilla.com/~bsterne/content-security-policy/origin-header-proposal.html*.
7. A. van Kesteren, "The From-Origin Header," (working draft) World Wide Web Consortium (July 21, 2011), *http://www.w3.org/TR/2011/WD-from-origin-20110721/*.
8. A. Barth, "The Web Origin Concept (v. 9)," IETF Draft (November 26, 2010), *http://tools.ietf.org/html/draft-abarth-origin-09*.
9. B. Sterne, "Content Security Policy" (2008), *http://people.mozilla.com/~bsterne/content-security-policy/*.
10. B. Sterne, "Content Security Policy," (draft) World Wide Web Consortium (March 15, 2011), *https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-unofficial-draft-20110315.html*.
11. I. Hickson, "HTML Living Standard," WHATWG (2011), *http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox*.
12. J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," (draft) IETF Request for Comments (August 5, 2011), *http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-02*.
13. A. Klein, "Google Chrome 6.0 and Above: Math.random Vulnerability" (2010), *http://www.trusteer.com/sites/default/files/Google_Chrome_6.0_and_7.0_Math.random_vulnerability.pdf*.
14. ".NET Framework 3.0: toStaticHTML Method," Microsoft, *http://msdn.microsoft.com/en-us/library/cc848922%28v=vs.85%29.aspx*.
15. D. Ross, "IE8 Security Part IV: The XSS Filter," *IEBlog* (2008), *http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx*.
16. E. Vela Nava and D. Lindsay, "Abusing Internet Explorer 8's XSS Filters" (2009), *http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf*.

Chapter 17

1. "navigator.registerProtocolHandler," Mozilla Developer Network, *https://developer.mozilla.org/en/DOM/window.navigator.registerProtocolHandler*.
2. "Manipulating the Browser History," Mozilla Developer Network, *https://developer.mozilla.org/en/DOM/Manipulating_the_browser_history/*.
3. A. Langley and M. Belsche, "SPDY: An Experimental Protocol for a Faster Web," The Chromium Projects, *http://www.chromium.org/spdy/spdy-whitepaper/*.
4. I. Fette and A. Melnikov, "The WebSocket Protocol," IETF Request for Comments draft (2011), *http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-10/* .
5. J. Rosenberg, M. Kaufman, M. Hiie, and F. Audet, "An Architectural Framework for Browser Based Real-Time Communications," IETF Request for Comments draft (2011), *http://tools.ietf.org/html/draft-rosenberg-rtcweb-framework-00/*.
6. I. Hickson, "HTML5: 5.6—Offline Web Applications," World Wide Web Consortium (2011), *http://www.w3.org/TR/html5/offline.html*.
7. A. Barth, "Simple HTTP State Management Mechanism," IETF Request for Comments draft (2010), *http://tools.ietf.org/html/draft-abarth-cake-00/*.
8. I. Hickson, "Web SQL Database: W3C Working Group Note 18," World Wide Web Consortium (2010), *http://www.w3.org/TR/webdatabase/*.
9. N. Mehta, J. Sicking, E. Graff, A. Popescu, and J. Orlow, "Indexed Database API: W3C Working Draft 19," World Wide Web Consortium (2011), *http://www.w3.org/TR/IndexedDB/*.
10. I. Hickson, "Web Applications 1.0: Web Workers," WHATWG (2011), *http://www.whatwg.org/specs/web-apps/current-work/complete/workers.html*.
11. A. Popescu, "Geolocation API Specification: Editor's Draft," World Wide Web Consortium (February 10, 2010), *http://dev.w3.org/geo/api/spec-source.html*.
12. "Detecting Device Orientation," Mozilla Developer Network, *https://developer.mozilla.org/en/detecting_device_orientation/*.
13. L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," Usenix HOTSEC (2011), *http://www.usenix.org/event/hotsec11/tech/final_files/Cai.pdf*.
14. "Web Developer's Guide to Prerendering in Chrome," Google code labs, *http://code.google.com/chrome/whitepapers/prerender.html*.
15. Z. Wang, "Navigation Timing: Editor's Draft," World Wide Web Consortium (July 27, 2011), *https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html*.

16. J. Gregg, "Web Notifications Overview: W3C Editor's Draft," World Wide Web Consortium (October 12, 2010), *http://dev.w3.org/2006/webapi/WebNotifications/publish/*.

17. D.D. Tran, I. Oksanen, and I. Kliche, "The Media Capture API: W3C Working Draft," World Wide Web Consortium (September 28, 2010), *http://www.w3.org/TR/media-capture-api/*.

INDEX

Symbols & Numbers

& (ampersand), in HTML, 71
< > (angle brackets)
  browser interpretation, 74–75
  in HTML, 71
<![CDATA[...]]> blocks, 72, 78, 250
<!DOCTYPE> directive, 71
<!ENTITY> directive, 76
<!– and –>, for HTML comments, 72
<% ... %> blocks, Internet Explorer and, 75
@ directives, in CSS, 89–90
\ (backslashes) in URLs, browser acceptance of, 29
` (backticks), as quote characters, 74, 111
!- directives, 76
// fixed string, in URLs, 25
% (percent sign), for character encoding, 31
. (period), hostnames with, and cookie-setting algorithms, 159
?-directives, 76
<?xml-stylesheet href=... ?> directive, 88
; (semicolon), as delimiter
  in HTTP headers, 48–49
  in URLs, 29
200–299 status codes, 54
300–399 status codes, 55
400–499 status codes, 55–56
500–599 status codes, 56

A

<a href=...> tag (HTML), 79
  target parameter, 174–175
about:blank document, origin inheritance, 165, 166–167
about:config (Firefox), navigation risks, 188
absolute URLs, vs. relative, 25
Accept-Language request header, 43
Accept request header, 43
Access-Control-Allow-Origin header, 237–238, 240
acrobat: scheme, 36
action parameter, for *<form>* tag, 80
ActionScript, 132–134
Active Server Pages, 75
ActiveX, 129, 136–137
address bars, 220
  and EV SSL, 65
  hiding, 221
  manipulation, 256–257
Adobe Flash, 119, 130, 132–134
  and cross-domain HTTP headers, 147n
  file handling without *Content-Type*, 199
  HTML parser offered by plug-in, 133
  policy file spoofing risks, 156–157
  security rules, 154–157
Adobe Reader, 130
Adobe Shockwave Player, 132
ADS (Alternate Data Stream) Zone Identifier, 231
advertisements, new window for, 217
Akamai Download Manager, 137
Allow-forms keyword, for *sandbox* parameter, 246
AllowFullScreen parameter, for Flash, 155
AllowNetworking parameter, for Flash, 155
Allow-same-origin keyword, for *sandbox* parameter, 246
AllowScriptAccess parameter, for Flash, 154
Allow-scripts keyword, for *sandbox* parameter, 246
Allow-top-navigation keyword, for *sandbox* parameter, 246
Alternate Data Stream (ADS) Zone Identifier, 231
ambient authority, 60, 60n
ampersand (&), in HTML, 71
anchor element (HTML), specifying name of, 28

angle brackets (< >)
  browser interpretation, 74–75
  in HTML, 71
anonymity, scripts and, 249
anonymous requests, in CORS, 239
anonymous windows, 175
antimalware, 236n
Apache
  and *Host* headers, 47
  PATH_INFO, 201
APNG file format, 83
Apple QuickTime, 119, 130, 132
Apple Safari. *See* Safari (Apple)
<*applet*> tag (HTML), 83, 128, 135, 183
application/binary, 212
application/javascript document type, 118
application/json document type, 118, 202
application/mathml+xml document type, 119
application/octet-stream document type, 200–201, 212
application/x-www-for-urlencoded, 81
Arce, Ivan, 2n
Arya, Abhishek, 209
asynchronous *XMLHttpRequest*, 146
Atom, 123
<*audio*> tag (HTML), 84, 119
authentication, in HTTP, 62–63
authorization, vs. authentication, 62n
Authorization header (HTTP), 63

B

background parameter for HTML tags, 83
background processes, in JavaScript, 258
backslashes (\) in URLs, browser acceptance of, 29
backticks (`), as quote characters, 74, 111
Bad Request status error (400), 55
bandwidth, and XML, 123n
Barth, Adam, 16, 177, 240, 241, 246, 257
Base64 encoding, 50n
basic credential-passing method, 63
Bell-La Padula security model, 2, 4
Berners-Lee, Tim, 9, 41, 69
  and semantic web, 72–73
  World Wide Web browser, 9
  World Wide Web Consortium, 11
<*bgsound*> tag (HTML), 84, 119
binary HTTP, 257
bitmap images, browser recognition of, 118
blacklists
  of HTTP headers in *XMLHttpRequest*, 147
  malicious URLs, 236n
_blank, as link target, 80
BMP file format, 83
<*body*> tag (HTML), 83
BOM (byte order marks), 208
Breckman, John, 52n
browser cache
  information in, 59
  poisoning, 60
browser extensions and UI, 161
browser-managed site permissions, 226–227
browser market share, May 2011, 19
browser-side scripts, 95–116
browser wars, 10–11, 233
buffer overflow, 265
bugs, preventing classes of, 7
Bush, Vannevar, 8
byte order marks (BOM), 208

C

cache. *See* browser cache
Cache-Control directive, 48, 59
cache manifests, 257
cache poisoning, 189, 263
caching behavior, in HTTP, 58–60
caching HTTP proxy, keepalive sessions and, 57
Caja, 116
Cake (proposal), 257
call stack, limiting size, 216
callto: scheme, 36
<*canvas*> tag (HTML5), 183
CAPTCHA, 184–185, 185n
Cascading Style Sheets (CSS), 11, 12, 73, 83, 87–93
  basic syntax, 88–90
  character encoding, 91–92
  interaction with HTML, 90
  opacity property, 179
  parser resynchronization risks, 90–91
  property definitions, 89
case of tags, HTML vs. XML, 72
<*![CDATA[…]]*> blocks, 72, 78, 250
certificate authorities, 64
certificates
  extended validation, 65
  warning dialog example, 66
cf: scheme, 36
characters
  delimiting, in URLs, 29
  encoding in CSS, 91–92
  encoding in filenames, 49–51
  encoding in HTML, 76–78
  encoding in JavaScript, 112–113
  encoding in URLs, 31–35
  printable, browser treatment of, 32

reserved, 31–35
unreserved, 32
character sets
byte order marks and detection, 208
detection for non-HTTP files, 210–211
handling, 206–211
for headers, 49–51
inheritance and override, 209
markup-controlled, on subresources, 209–210
sniffing, 264
in URLs, 33
@charset (CSS), 89
children objects in JavaScript, 108
Chrome
autodetection of passive document types, 205
cached pages in, 37
characters in URL scheme name ignored by, 25
deleting JavaScript function, 103
and file extensions in URLs, 130
local file access, 160
modal dialogs for prompts, 219
navigation timing, 259
prerendering page, 258–259
printable characters in, 32
privileged JavaScript in, 161
and realm string, 63
and RFC 2047 encoding, 50
stored password retrieval, 228
SWF file handling without *Content-Type*, 199
time limits on continuously executing scripts, 215
WebKit parsing engine, 70n
window.open() function and, 218
Windows Presentation Foundation plug-ins, 136
chunked data transfers, 57–58
clickjacking, 179, 180–181, 263
click() method, 218
client certificates, 64–66
client-server architecture, 17–18
client-side data, 165
client-side databases, 258
client-side errors (400–499), 55–56
client-side scripts, restricting privileges of HTML generated by, 250–251
cloud, 15
Clover, Andrew, 184
command injection, 265
comments
in CSS syntax, 89
in XHTML and HTML, 72
Common UNIX Printing System (CUPS), 152–153
Common Vulnerability Scoring System (CVSS), 6–7
Common Weakness Enumeration (CWE), 6
complex selectors, in CSS, 88
computer proficiency of user, 14
conditionals, explicit and implicit, in HTML, 75–76
conflicting headers, resolution of, 47–48
CONNECT requests, 46, 54
Connolly, Dan, 9
content directives, on subresources, 204
Content-Disposition directive, 48, 84, 122
defensive uses, 203–204
NUL character and, 51
plug-in-executed code and, 204
user-controlled filenames in, 67
content inclusion in HTML
hyperlinking and, 79–84
type-specific, 82–84
Content-Length header, 43, 52, 147
in keepalive sessions, 56–58
content recognition, 197–211
content rendering, plug-ins for, 127–138
Content Security Policy (CSP), 242–245, 250, 253
criticisms of, 244–245
violations, 244
content sniffing, 197–198, 205, 264
Content-Type directive, 49, 71, 84
application/binary, 212
application/JavaScript, 118
application/json, 118, 202
application/mathml+xml, 119
application/octet-stream, 200–201, 212
charset parameter, 206, 208
image/jpeg, 118, 202, 205
image/svg+xml, 124
logic to handle absence, 198–199
plug-ins and, 128, 204
slash-delmited alphanumeric tokens in, 199
special values, 200–201
text/css, 118
text/html, 124
text/plain, 118, 156, 200–201, 204, 212
unrecognized, 202–203
and XML document parsing, 120
control characters, JavaScript shorthand notation, 112
cookie-authenticated text, reading, 181
Cookie header. *See* cookies
cookie injection, 264

cookies, 11, 257
  deleting, 62
  and DNS hijacking, 153
  forcing, 264
  limitations on third-party, 192–194
  and same-origin policy, 150–151
  security policy for, 149–153
  semantics, 60–62
  user data in, 67
CORS. *See* Cross-Origin Resource Sharing (CORS)
CR characters, stripping from HTTP headers, 45
credential-passing methods, 63
credentials, in URLs, 26
CRLF (newline), 45
cross-browser interactions, 16–17
cross-document links, 8, 9
cross-domain communications, and frame descendant policy, 176–178
cross-domain content inclusion, 181–183
cross-domain policy files, 155–156
cross-domain requests, 236–239
Cross-Origin Resource Sharing (CORS), 148, 236
  current status, 239
  non-simple requests and preflight, 238
  request types, 236–237
  security checks, 237–238
cross-origin subresources, 183
cross-site request forgery (XSRF, CSRF), 84, 190, 262
  exploitation of flaws, 190
  login forms and, 145–146
cross-site script inclusion (XSSI), 104n, 262
cross-site scripting (XSS), 71, 262
  bugs, and password managers, 228
  exploitation of flaws, 190
  filtering, 251–252, 253
crossdomain.xml file, 155, 162
CSP (Content Security Policy), 242–245, 250, 253
CSRF (cross-site request forgery), 84, 190, 262
  exploitation of flaws, 190
  login forms and, 145–146
CSS. *See* Cascading Style Sheets (CSS)
CUPS (Common UNIX Printing System), 152–153
currentStyle API, 184
CVSS (Common Vulnerability Scoring System), 6–7
CWE (Common Weakness Enumeration), 6
Cyrillic alphabet, homoglyphs in, 35

D

daap: scheme, 36
data: scheme, 37, 167–168
data transfers, chunked, 57–58
Date/If-Modified-Since header pair, 59
deceptive framing, 180
dedicated workers, for background processes, 258
default policy, CSP directive for, 243
default ports, for protocols, overriding, 27
DELETE method (HTTP), 53
deleting
  cookies, 62
  JavaScript functions, 102–103
delimiting characters, in URLs, 29
denial-of-service (DoS) attacks, 214–219, 248, 264
DeviceOrientation API, 258
dialog use restrictions, 218–219
digest credential-passing method, 63
Digital Rights Management (DRM), 131
directory traversal, 265
disable-xss-protection, 242n
<*div*> tag (HTML), 73
DNS hijacking, and cookies, 153
DNS labels, security mechanisms based on, 142n
DNS names, in URLs, browser acceptance, 27
DNS pinning, 142n, 190
DNS rebinding, 142n, 189
DNT request header, 193
<*!DOCTYPE*> directive, 71
document.cookie API (JavaScript), 61
document.domain property (JavaScript), 143–144
document-level scrollbar, 180
document namespace, mapping HTML elements to, 110
document object (JavaScript), 108
Document Object Model, 12, 108, 109–111, 142–146
document rendering helpers, 130–131
documents
  changing location of existing, 174–178
  script access to other, 111–112
document type detection logic, 198–206
Domain parameter, for cookie, 61
domains
  hardcoded, 227
  problems with restrictions, 151–152
DOMService mechanism, 158
DoS (denial-of-service) attacks, 214–219, 248, 264

downloaded files, 205–206
drag-and-drop, 180
DRM (Digital Rights Management), 131
duplicate headers, resolution of, 47–48
Dutta, Sunava, 239

E

E4X. *See* ECMAScript for XML (E4X)
Earthlink, 153
ECMA (European Computer Manufacturers Association), 11, 96
ECMAScript, 96
  escape codes, 112
  strict mode, 104
ECMAScript for XML (E4X), 106–107
Eich, Brendan, 95
Electronic Frontier Foundation, 109
Eloquent JavaScript (Haverbeke), 97
<embed> tag (HTML), 83
  mixed content, 183
  src=..., 128
EMF file format, 83
encapsulating pseudo-protocols, 37–38
encoding schemes, for headers, 49–51
encryption, protocol-level, 64–66
enctype="text/plain", for *<form>* tag, 81
endless loop, 101, 215
ENQUIRE, 9, 10
<!ENTITY> directive, 76
entity encoding, in HTML, 76–78
error-handling rules, for certificates, 65–66
escaping reserved characters, in HTML, 71
escaping scheme, 91
Esser, Stefan, 209
ETag/If-None-Match header pair, 59
European Computer Manufacturers Association (ECMA), 11, 96
eval() function, 102
eval-script, 242n
Evans, Chris, 181, 182
EV SSL (Extended Validation SSL), 65
exception
  for *eval()* function, 102
  recovery in JavaScript, 100
execution time for scripts, 215–216
Expires directive, 48, 59
Expires parameter, for cookie, 61
explicit conditionals, in HTML, 75–76
expression(...) function (CSS), 89
Extended Validation SSL (EV SSL), 65
Extensible Application Markup Language (XAML), 134
extension matching, 202n
ExternalInterface.call() API, 133
External XML Entity (XXE) attack, 76

F

false positives, risk in XSS filtering, 251–252
fault tolerance, 11
feeds, 123–124
feed: scheme, 37
Felten, Ed, 193
file extensions, browser response to, 205
file formats. *See also* plug-ins
  audio and video, 119
  bitmap images, 118
  HTML. *See* HTML
  non-renderable, 124
  plaintext, 64, 85, 117–118
  XML. *See* XML
file inclusion, 265
file path, hierarchical, in URLs, 27–28
file: protocol, 159–160, 188
files, downloaded, 205–206
File Transfer Protocol (FTP), 26n, 205–206
filtering
  pop-up, 217–218
  reserved characters, in HTML, 71
Firefox (Mozilla), 13, 17
  and ActiveX, 137
  cached pages in, 37
  character set inheritance, 209
  CORS in, 239
  and credential portion of URLs, 26
  data: URLs in, 168
  DNT request header, 193
  entity names, 77
  external content directives, 90
  Gecko parsing engine, 70n
  history.pushState() API, 256
  javascript: URLs in, 169
  local file access, 160
  modal dialogs for prompts, 219
  multiple cookies for, 62
  printable characters in, 32
  privileged JavaScript in, 161
  prompt displayed when saving *Content-Type: image/jpeg* document, 205
  redirects to *about:blank*, 166
  and RFC 2047 encoding, 50
  RSS and Atom renderers for, 124
  same-origin policy loopholes, 185
  stored password retrieval, 228
  Strict Transport Security support, 248
  SWF file handling without *Content-Type*, 199

Firefox (*continued*)
time limits on continuously executing scripts, 215
UTF-8 text in, 50
Windows Presentation Foundation plug-ins, 136
Worker API, 258
firefoxurl: protocol, 17, 36
Flash applets, 11
fonts
CSP directive for, 243
Flash programs enumeration of, 132
Forbidden status code (403), 56
forecasting, statistical, 6
format-string vulnerability, 266
form-based password managers, 227–229
form feed character, in HTML tag, 74
forms, 80–82
Found status code (302), 55
fragment ID, in URLs, 28–29
frame-ancestors directive, 243
framebusting, 264
frame descendant policy, and cross-domain communications, 176–178
frames, 82
disabling navigation descendant model, 230–231
hijacking risks, 175–176
name attribute of, 175
sandboxed, 245–247
unsolicited, 178–181
and window interactions, 174–181
frame-src directive, 243
From-Origin header, 240
FTP (File Transfer Protocol), 26n, 205–206
ftp: scheme, 36
full-screen mode, proposals for, 259
fully qualified absolute URLs, 24
fully restricted URL scheme, 188
functional notation, in CSS, 89
functions
JavaScript, overriding, 102–103
resolution for JavaScript, 98–99

G

Gabrilovich, Evgeniy, 35
Gecko parsing engine, 70n
Generalized Markup Language (GML), 8–9
geolocation data, 226
geolocation discovery, 258
geolocation-sharing prompts, 223
getComputedStyle API, 184
getElementById() function, 109
getElementsByTagName() function, 109
GET method (HTTP), 42, 52, 58, 80–81
GetRight download utility, 137
getters, in JavaScript, 103
getURL() function, 133
GIFAR vulnerability, 129
GIF file format, 83, 129
GML (Generalized Markup Language), 8–9
Gontmakher, Alex, 35
Gonzalez, Albert, 5n
gopher: scheme, 36
Gosling, James, 134
GPS data, 226n
Grossman, Jeremiah, 179
Guninski, Georgi, 176

H

Hansen, Robert, 179
hardcoded domains, 227
Haverbeke, Marijn, *Eloquent JavaScript*, 97
HDP file format, 83
header injection, 45, 239, 262
headers
character set and encoding schemes, 49–51
Content Security Policy encoded in, 242
in HTTP requests, 43
resolution of duplicate or conflicting, 47–48
semicolon-delimited values, 48–49
HEAD request (HTTP), 53
hexadecimal notation, 77, 112
hierarchical file path, in URLs, 27–28
history object (JavaScript), 108
history.pushState() API, 256
Hodges, Jeff, 248
homoglyphs, in Cyrillic alphabet, 35
Host request header, 43
hostnames
extra periods, and cookie-setting algorithms, 159
non-fully qualified, 159
HTML (Hypertext Markup Language), 9, 69–86
basic concepts, 70–73
case of tags, 72
converting to plaintext, 85
CSS interaction with, 90
document misidentified as, 198
document parsing modes, 71–72
embedded in feed formats, 124
entity encoding, 76–78
explicit and implicit conditionals, 75–76

HTTP integration semantics, 78–79
hyperlinking and content inclusion, 79–84
in-browser sanitizers, 250–251
mapping elements to *document* namespace, 110
parser behavior, 73–76
tag interactions, 74–75
type-specific content inclusion, 82–84
version 4, 12
version 5, 70, 119, 131
HTTP (HyperText Transfer Protocol), 9, 41–67
authentication, 62–63
basic syntax, 42–51
binary, 257
caching behavior, 58–60
cookie semantics, 60–62
downgrade, 264
history, 41–42
HTML integration semantics, 78–79
newline handling, 45
proxy requests, 46–47
request types, 52–54
semantics battle, 72–73
simultaneous connections, 216
version 0.9, 42–43, 44
version 1.0, 42, 43, 44, 48, 59
version 1.1, 42–43, 45, 48, 57, 198
httponly flag, for cookie, 61, 150
http: scheme, 36
HTTPS, 65
documents, 138n, 183
downgrade risks, 248
https: scheme, 36
hyperlinking, and content inclusion, 79–84
Hypertext Markup Language (HTML). *See* HTML (Hypertext Markup Language)
HyperText Transfer Protocol (HTTP). *See* HTTP (HyperText Transfer Protocol)

I

IANA (Internet Assigned Numbers Authority), 24, 152
ICO file format, 83
IDNA (Internationalized Domain Names in Applications), 34–35
IETF (Internet Engineering Task Force), 11
If-Modified-Since header, 59
If-None-Match header, 59
<*iframe*> tag (HTML), 82, 176, 209, 245–247
image/jpeg document type, 118, 202, 205
images
bitmap, 118
in HTML, 83
risk of content sniffing on, 202
Scalable Vector Graphics (SVG), 83, 121–122
image/svg+xml document type, 124
<*img*> tag (HTML), 83
src parameter, 181
for SVG images, 122
implicit caching, 59
implicit conditionals, in HTML, 75–76
@import, in CSS, 89–90
IndexedDB design, 258
indicator of hierarchical URLs, 25–26
information security, 1–8
inheritance, for *vbscript:* scheme, 169–170
inline-script setting, 242n
innerHTML property, 110–111
innerStaticHTML API, 251
integer overflow, 266
Interactive Voice Response (IVR) systems, 236
interconnected systems, losses in, 5
internal networks, access to, 189–190
Internal Revenue Service, 231
Internal Server Error (500), 56
International Organization for Standardization (ISO), 11
Internationalized Domain Names in Applications (IDNA), 34–35
Internet Assigned Numbers Authority (IANA), 24, 152
Internet Engineering Task Force (IETF), 11
Internet Explorer, 10, 11–12
ActiveX and, 137
and <% ... %> blocks, 75
\ (backslash) in URLs, 29
acceptance of backtick as quote, 74
characters in URL scheme name ignored by, 25
clickjacking, 182
content sniffing, 202
cookies, 149
data: URLs in, 168
delete attempt of JavaScript function, 103
extension matching, 202
fallback display, 118
and file extensions in URLs, 130
frames, 177
JavaScript in, 96
JSON.parse() function alternative, 104
local file access, 160
markup controlled charset on, 209

Internet Explorer (*continued*)
and multiline headers, 45
multiline string literals support, 91
non-recognition of vertical tab, 112
NUL character and, 73, 74
origin check and port number, 142
printable characters in, 32
proprietary *security-restricted* parameter, 246
redirects to *about:blank*, 166–167
and RFC 2047 encoding, 50
same-origin policy and, 143n, 185
Silverlight and, 134
stored password retrieval, 228
SWF file handling without *Content-Type*, 199
text/plain document type, 200–201
third-party cookies blocking, 193
time limits on continuously executing scripts, 215
Trident parsing engine, 70n
VBScript, 96, 114
window.open() function and, 218
Windows Presentation Foundation plug-ins, 136
XDomainRequest approach to, 148
XSS-detection logic, 251
Zone.Identifier metadata, 231
zone model, 229–231
Internet Information Server, and *Host* headers, 47
Internet service providers, 153
Internet zone, for Internet Explorer, 230
interstitials, 218
intrusions
escalation of, 5
nonmonetary costs, 5
Invisible Gorilla experiment, 223
IP addresses, and cookies, 158
ISO (International Organization for Standardization), 11
ISO-8859-1 (Western European code page), 50
itms: scheme, 36
itpc: scheme, 36
IVR (Interactive Voice Response) systems, 236

J

Jackson, Collin, 16, 177, 184, 240
jar: scheme, 37
Java, 134–135, 157–158
Java Runtime Environment (JRE), 135
JavaScript, 10, 11n, 83, 95–107
character encoding in, 112–113
code and object inspection capabilities, 101–102
code execution, 100
code inclusion modes and nesting risks, 113–114
document.domain property, 143–144
Document Object Model, 12, 108, 109–111
embedded in PDF documents, 130
execution order control, 100–101
labeled statements support, 105n
MIME type, 118n
Netscape and, 95–96
runtime environment for, 102–104
script processing model, 97–100
setters and getters, 103
standard object hierarchy, 107–112
variable declaration, 99
and WML Script (WMLS), 123
JavaScript Object Notation (JSON), 104–106, 112
javascript: scheme, 37, 169–170
Jobs, Steve, 131
JPEG file format, 83
JScript, 11n
JScript.Encode, 113n
JSObject mechanism, 158
JSON (JavaScript Object Notation), 104–106, 112
JSONP (JSON with padding), 106n, 245
JSON.parse() function, alternatives, 104

K

Kaminsky, Dan, 153
katakana, 33
keepalive sessions, 56–57, 216
keystroke redirection, 180
Kinugawa, Masato, 210

L

language parameter, for *<script>* tag, 113n
Lessig, Lawrence, 192
LF (newline), HTTP quirks in handling, 45
LFI (local file inclusion), 265
Lie, Wium, Håkon, 87
<link rel=stylesheet> directive, 88
<link rel=stylesheet href=...> tag, 181
LiveScript, 95
livescript: scheme, 37

loadPolicyFile() method, 155–156
local file inclusion (LFI), 265
local files, access issues, 159–160
local intranet zone, for Internet Explorer, 229
local machine zone, for Internet Explorer, 229
localhost, danger of, 152–153
localStorage object (JavaScript), 148
location.hash, 256
location headers, sending user-controlled, 67
location.host property, 173
location object (JavaScript), 108, 153–154
location of documents, changing, 174–178
login forms, autocompletion by browsers, 228
lookup functions, in Document Object Model, 109
loopback interfaces, 152n
Lynx, 10

M

Macromedia Flash, 132
mailto: protocol, 25, 36, 256
mail user agent (MUA), 203n
malicious sites, blacklist-driven attempts to block, 226
managed code, 134n
Mark of the Web (MotW), 204, 231
markup filter for user content, 86
mashups, 176
MathML (Mathematical Markup Language), 72, 122
Math.random() function, 109
max-age parameter
  for cookie, 61
  for STS record, 248
media capture, 259
Memex, 8
memory pointers, 266
memory use restrictions for scripts, 215–216
<*meta*> directive, 206, 208
<*meta http-equiv*=> directive, 78–79
meta-policies, for Flash, 156–157
mhtml protocol, 38
Microsoft. *See also* Internet Explorer
  descendant policy development, 177
  .NET Framework with XPAB plug-ins, 136
  objections to CORS, 239
  Sun suit over Java virtual machine, 135n
  Threats Against and Protection of Microsoft's Internal Network, 5n
  Windows operating system, 10
Microsoft Office, 130
Microsoft Silverlight, 119, 134, 157
MIME (Multipurpose Internet Mail Extensions), 43n, 81n
  malformed types, 199
  mapping types to plaintext, 118
  for plug-ins, 128
  specialized for content in sandboxed frame, 247
Mitchell, John C., 177, 240
mixed content, 183, 262–263
mmst: scheme, 36
mmsu: scheme, 36
Mocha language, 95
mocha: scheme, 37
modal behavior of dialogs, 218–219
Montulli, Lou, 60
Mosaic, 10. *See also* Netscape
MotW (Mark of the Web), 204, 231
mouse cursors, redefining, 89n
Moved Permanently status code (301), 55
Mozilla Firefox. *See* Firefox (Mozilla)
Mozilla specification, 242
msbd: scheme, 36
MsgBox (VBScript), 114
MUA (mail user agent), 203n
multiline headers, support for, 45
multiline string literals
  Internet Explorer support, 91
  in JavaScript, 113
multimedia playback, 130
Multipurpose Internet Mail Extensions (MIME). *See* MIME (Multipurpose Internet Mail Extensions)
My computer zone, for Internet Explorer, 229

N

name attribute, of frames, 175
named entities, 76
namespace in JavaScript, 107
name: value pairs, in HTTP requests, 43
name=value pairs
  cookies for storing, 60
  for forms, 81
National Science Foundation, backbone network, 10
Naval Research Laboratory, 3–4
navigateToURL() function, 133

navigation
to sensitive schemes, 188
timing, 259
navigator.device.capture API, 259
navigator.geolocation.getCurrentPosition() API, 258
navigator object (JavaScript), 108
navigator:registerProtocolHandler() API, 256
Negotiate authentication method, 63
.NET runtime, 135
Netflix, 134
Netscape
cookie specification, 151–152
and JavaScript, 95–96
and same-origin policy, 142
Netscape Navigator, 11
network fenceposts, 264
networking, HTTP-less, 257
New York Times, 192
newline, HTTP quirks in handling, 45
news: scheme, 36
NLS, 9
nntp: scheme, 36
no-cache value, for *Cache-Control* header, 59
No Content status code (204), 54
noncanonical encodings, 32n
nonencapsulating pseudo-protocols, 37
non-HTTP resources
character set detection for, 210–211
proxies allowing requests for, 46
non-renderable file types, 124
non-US-ASCII text, in URLs, 32–35
no-store value, for *Cache-Control* header, 59
Not Found status code (404), 56
Not Modified status code (304), 55, 59
Notification API, 259
NTLM authentication method, 63
NUL character, and HTTP headers, 51
NUL-containing strings, JavaScript and, 109

O

Object Linking and Embedding (OLE), 136
<*object*> tag, 83, 84
data=..., 128
mixed content, 183
octal character codes, JavaScript support, 112
Ogg Theora, 119
OK status code (200), 54
OLE (Object Linking and Embedding), 136
onbeforeunload dialog, 219n
onerror handler, on <*img*> tag, 184
onerror parameter, 74
onkeydown event (JavaScript), 180
onload handler, to measure load time for document, 184
onmousemove events, 222
opacity property (CSS2), and JavaScript code, 179
opener.window.focus() function, 217n
OpenGL-based 3D graphics, 131n
open redirection, 263
Opera, 10
data: URLs in, 168
deleting JavaScript function, 103
and file extensions in URLs, 130
history.pushState() API, 256
local file access, 160
modal dialogs for prompts, 219
and multiline headers, 45
period-counting problem in, 159
Presto parsing engine, 70n
printable characters in, 32
redirects to *about:blank*, 167
Refresh redirection to *javascript:*, 170
and RFC 2047 encoding, 50
RSS and Atom renderers for, 124
stored password retrieval, 228
SWF file handling without *Content-Type*, 199
Worker API, 258
OPTIONS method (HTTP), 53
Origin header, 240–241
origin inheritance, 165–171
about:blank document, 166–167
for *javascript:* scheme, 169–170
origins
ambiguous or unexpected, 158–161
attempts to broaden, 143
Ormandy, Tavis, 152
outerHTML property, 110–111
overwriting cookie, 62

P

P2P networking, 257
P3P (Platform for Privacy Preference), 193
Panopticlick, 109
parallel HTTP connection design, 216
<*param*> tag (HTML), for plug-ins, 128
_parent, as link target, 80
parsing
behavior fundamentals, 73–76
JavaScript, 97–98
modes for HTML documents, 71–72
resynchronization risks, 90–91

parsing engines, for browsers, 70n
Partial Content status code (206), 54
partly restricted URL scheme, 188
passive multimedia, CSP directive for, 243
password
  in credentials portion of URLs, 26
  form-based managers, 227–229
  methods for passing, 63
Path parameter, for cookie, 61
path value, for cookie, 149–150
payload inspection, by Internet Explorer, 202
PDF documents130–131
percent encoding, 31
percent sign (%), for character encoding, 31
per-host connection limit, 216
period (.), hostnames with, and cookie-setting algorithms, 159
permissions, browser- and plug-in-managed, 226–227
permitted-cross-domain-policies parameter, for *crossdomain.xml* file, 162
persistent workers, for background processes, 258
Petkov, Petko D., 131
phishing, 176n
plaintext
  converting HTML to, 85
  as file format, 117–118
  for HTTP session information, 64
<plaintext> tag (HTML), 72
Platform for Privacy Preference (P3P), 193
plug-ins, 10–11
  ActiveX, 129, 136–137
  Adobe Flash. *See* Adobe Flash
  application frameworks as basis, 131–136
  content, 83
  for content rendering, 127–138
  CSP directive for, 243
  document rendering helpers, 130–131
  invoking, 128–130
  Microsoft Silverlight, 119, 134, 157
  for PDF documents, 130–131
  perils of content-type handling, 129–130
  protocols claimed by, 36–37
  security rules, 153–158
  site permissions management, 226–227
  Sun Java, 134–135, 157–158
  XML browser applications (XBAP), 135–136
PNG file format, 83
pointers, management vulnerabilities, 266
poisoned browser cache, on trusted network, 60
pop-under, 217
pop-up filtering, 217–218
ports
  default, for protocols, overriding, 27
  prohibited, 190–192
positioning windows, 219–222
postMessage(...) API, 144–145, 258
POST method (HTTP), 52, 81
postponing JavaScript execution, 101
Pragma: no-cache request header, 59
prerendering web page, 258–259
presentation, HTML tags for, 73
PresentationHost.exe, 135
pressed key, examining code of, 180
Presto parsing engine, 70n
printable characters, browser treatment of, 32
privacy-related side channels, 184–185
private browsing modes, 249, 253
private value, for *Cache-Control* header, 59
privileges, site, 225–234
prohibited ports, 190–192
properties, definitions in CSS, 89
proposals
  content-level, 258–259
  I/O interfaces, 259
  URL- and protocol-level, 256–257
protocol-host-port tuple, 142, 241
protocol-level information
  encryption, 64–66
  preserving, 78
protocol-level proposals, 256–257
protocols
  claimed by third-party applications, 36–37
  default ports for, overriding, 27
  registration, 256
  in URL scheme name, 24
proxy-originating error responses, browser processing, 47
proxy requests, 46–47
pseudo-functions (CSS), 89
pseudo-protocols
  encapsulating, 37–38
  nonencapsulating, 37
pseudo-URLs, 23, 24, 165
  restricted, 170–171
  and same-origin policy, 161
public key cryptography, 64, 64n
Public Suffix List, 159
public value, for *Cache-Control* header, 59
public Wi-Fi networks, and HTTP caching risk, 60
Punycode, 34
purging browser cache, 60
PUT request (HTTP), 53

Q

query string in URLs, 28
QuickTime (Apple), 119, 130, 132
quote characters, in HTML, 71, 74
quoted-printable encoding scheme, 50n
quoted-string syntax, 48–49
  and cookies, 62
  for CSS property values, 89

R

race conditions, in JavaScript, 101
raw text, for CSS property values, 89
Really Simple Syndication (RSS), 123
realm string, 62
RealNetworks RealPlayer, 130, 132
redirect headers, sending user-controlled, 67
Redirection status codes (300–399), 55
Referer header, 43, 51
  alternative to, 240
  leakage, 263
relative URLs, 24
  vs. absolute, 25
  input filters, 40
  resolution of, 38–39
remote file inclusion (RFI), 265
Request for Comments (RFC). *See* RFC (Request for Comments)
request headers, in HTTP, 43
request types
  form-triggered, 80–82
  HTTP, 52–54
reserved characters, in HTML, 31–35, 71
resource exhaustion attacks, 214
response codes, server, 54–56
response splitting, 45
Restricted sites zone, for Internet Explorer, 229–230
revalidation, 59
RFC (Request for Comments)
  1630
    on query string format, 28
    on reference parser, 25–26
  1738, on URLs, 24, 25
  1866, on HTML 2.0, 69
  1945
    on HTTP, 42
    and TEXT token, 50
  2046, on *application/octet-stream*, 200
  2047, for non-ISO-8859-1 string format, 50
  2109, on cookies, 60, 61, 62
  2183, on *Content-Disposition* header, 203
  2368, on query string format, 28
  2616, 44
    on GET requests, 58
    on HTTP, 42
    on resolving ambiguities, 47
    status codes for server response, 54
    on URLs, 24
  2617, on authentication, 62
  2818, on encapsulation, 64
  2965, on *Cookie2*, 60
  3490, 34
  3492, 34
  3986, 24, 25, 33
  4627, on JSON, 104
  4918, on WebDAV, 54
  6265, on cookies, 61
  browser permissions to examine payload, 198
  on HTTP, 48
RFI (remote file inclusion), 265
rgb(...) pseudo functions (CSS), 89
Riley, Chris John, 203
Rios, Billy, 129
risk management, 4–6
root object in JavaScript, 107
Ross, David, 251
rotate(...) pseudo functions (CSS), 89
RSS (Really Simple Syndication), 123
rtsp: scheme, 36
runtime environment, for JavaScript, 102–104

S

Safari (Apple), 13
  and credential portion of URLs, 26
  deleting JavaScript function, 103
  hiding address bar, 221
  and multiline headers, 45
  and realm string, 63
  RSS and Atom renderers for, 124
  SOP bypass flaws, 142n
  stored password retrieval, 228
  SWF file handling without *Content-Type*, 199
  text/plain document type, 200–201
  third-party cookies, 193
  time limits on continuously executing scripts, 215
  WebKit parsing engine, 70n
safeInnerHTML API, 251
same-origin policy mechanism, 16
  cookies impact on, 150–151
  for Document Object Model, 142–146
  limitations, 173–186
  loopholes, 185

and pseudo-URLs, 161
for web storage, 148
for *XMLHttpRequest* API, 146–148
sandbox directive, 244
sandboxed frames, 245–247, 250, 253
scripting, forms and navigation restrictions, 247
synthetic origins, 247
sanitization
in-browser HTML, 250–251
of tags, 76
Scalable Vector Graphics (SVG), 83, 121–122
scale(...) pseudo functions (CSS), 89
schemes
current list of valid names, 24
input filters, 40
name in URLs, 24–25
navigation to sensitive, 188
Schwab, Charles, 230–231
screen object (JavaScript), 108
script-nonce directive, 244
scripts, 83
access to other documents, 111–112
browser-side, 95–116
connection limits, 216–217
dialog use restrictions, 218–219
execution time and memory use restrictions, 215–216
pop-up filtering, 217–218
rogue, 213–224
specifying charset, 209
script-src directive (CSP), 242
<*script*> tag (HTML), 72
JSON and, 104–105
language parameter, 113n
parsing and, 98
src parameter, 181
<*script*> tag (XHTML), 78
scrollbar, document-level, 180
Secure attribute, for cookie, 61
secure cookies, 150, 162
security
actions subject to checks, 141
definition, 2–4
new and upcoming features, 235–253
practical approaches, 7–8
quality assurance, 7
Security.allowDomain(...) method, for Flash, 155
security dialogs, attacks on, 222–223
security engineering cheat sheet
building web applications on internal networks, 195
Content Security Policy (CSP), 253
converting HTML to plaintext, 85
cross-domain communications in JavaScript, 162, 186
cross-domain resources, 186
cross-domain *XMLHttpRequest* (CORS), 253
data: and *javascript:* URLs, 172
decoding parameters received through URLs, 40
embedding plug-in-handled active content from third parties, 162
enabling plug-in-handled files, 138
filtering user-supplies CSS, 93
generating documents with partly attacker-controlled contents, 212
generating HTML documents with attacker-controlled bits, 85
good practices for all websites, 212
hosting user-generated files, 212
hosting XML-based document formats, 125
hosting your own plug-in-executed content, 163
hygiene for all HTML documents, 85
interacting with browser objects on client side, 115
launching non-HTTP services, 195
loading remote scripts, 115
loading remote stylesheets, 93
markup filter for user content, 86
non-HTML document types, 125
parsing JSON from server, 115
permitting user-created <*iframe*> gadgets on site, 224
private browsing modes, 253
putting attacker-controlled values into CSS, 93
relying on HTTP cookies for authentication, 162
requesting elevated permissions within web application, 232
sandboxed frames, 253
security hygiene for all websites, 186
security policy hygiene for all websites, 162
security-sensitive UIs, 224
sending user-controlled location headers, 67
sending user-controlled redirect headers, 67
serving plug-in-handled files, 138
Strict Transport Security, 253
third-party cookies for gadgets or sandboxed content, 195
toStaticHTML() API, 253

security engineering cheat sheet (*continued*)
URL input filters, 40
URLs constructed based on user input, 40
user-controlled filenames in *Content-Disposition* headers, 67
user-controlled scripts, 116
user data in HTTP cookies, 67
user-specified class values on HTML markup, 93
user-supplied data inside JavaScript blocks, 115
writing browser extensions, 163
writing plug-ins or extensions recognizing privileged origins, 232
XDomainRequest, 253
XSS filtering, 253
security model extension frameworks, 236–241
cross-domain requests, 236–239
XDomainRequest, 239–240
security model restriction frameworks, 241–249
See Other status code (303), 55
selector suffixes, in CSS, 88
_self, as link target, 80
self-closing tag syntax, 72
semantic web, 72–73
semicolon (;), as delimiter
in HTTP headers, 48–49
in URLs, 29
server address, in URLs, 26–27
server port, in URLs, 27
server response codes, 54–56
server-side code, common problems unique to, 265–266
server-side errors (500–599), 56
Service Unavailable error (503), 56
sessionStorage object (JavaScript), 148
Set-Cookie headers, 61
setters, in JavaScript, 103
SGML (Standard Generalized Markup Langauge), 9
shared workers, for background processes, 258
Shockwave Flash, 132
SHODAN, 203
showModalDialog() method, 217
shttp: scheme, 36
Simple Mail Transfer Protocol (SMTP), 27, 44, 190
sip: scheme, 36
<site-control permitted-cross-domain-policies=."..."> parameter, 157
site privileges, 225–234
browser- and plug-in-managed permissions, 226–227
skew(...) pseudo functions (CSS), 89
SMTP (Simple Mail Transfer Protocol), 27, 44, 190
social engineering attacks, 32n
software, difficulty analyzing behavior of, 3
** tag (HTML), 73
SPDY (Speedy), 257
Spyglass Mosaic, 10
SSL, warnings appearance, 66
Standard Generalized Markup Langauge (SGML), 9
statistical forecasting, 6
Sterne, Brandon, 242
Stone, Paul, 180
Strict Transport Security (STS), 248–249, 253
strict XML mode, 72
stylesheets
CSP directive for, 243
specifying charset, 209
<style> tag (HTML), 72
<style> tag (XHTML), 78
subframes, CSP directive for, 243
subresources
cross-origin, 183
markup-controlled charset on, 209–210
Sun Java, 134–135, 157–158
Sun Microsystems, 129
SVG (Scalable Vector Graphics), 83, 121–122
<svg> tag (HTML5), 122
synchronous *XMLHttpRequest*, 146
syntax-delimiting characters, in URLs, 31
"syntax error" message, retrieved file snippet in, 181

T

<table> tag (HTML), 83
tags, in HTML, 70
handling those not closed before end of file, 75
interactions, 74–75
sanitization, 76
target parameter, for *<a href=...>* tag (HTML), 79, 174–175
taxonomy, 6–7
TCP/IP, HTTP and, 42
TCP (Transmission Control Protocol), 42n
connections via XMLSocket, 156
list of prohibited ports, 190–192
Temporary Redirect status code (307), 55

testing, for Internet Explorer use, 112
text/css document type, 118
text/csv document type, 198
text/html document type, 124
text message, sending to window with valid JavaScript handle, 144
text/plain document type, 118, 156, 200–201, 204, 212
TEXT token, 50
<*textarea*> tag (HTML), 72, 111
third-party applications, protocols claimed by, 36–37
third-party cookies, limitations, 192–194
threat evolution, 14–18
  cloud, 15
  nonconvergence of visions, 15–16
  user as security flaw, 14–15
Threats Against and Protection of Microsoft's Internal Network (Microsoft), 5n
three-step TCP handshake, 56
TIFF file format, 83
timer, in JavaScript, 101
timing attacks, on user interfaces, 222–223
TLS (Transport Layer Security), 64
_top, as link target, 80
top-level domains, 152
toSource() method (JavaScript), 101
toStaticHTML() API, 250–251, 253
toString() method (JavaScript), 101
TRACE method (HTTP), 53
tracking, unscrupulous online, 193
tragedy of the commons dilemma, 3
Transfer-Encoding: chunked scheme, 58
Transmission Control Protocol (TCP). *See* TCP (Transmission Control Protocol)
Transport Layer Security (TLS), 64
Trident parsing engine, 70n
Trusted sites zone, for Internet Explorer, 229
Turing, Alan, 3n
type parameter, for plug-in tag, 128

U

UI spoofing attacks, and Flash, 132
unauthenticated requests, by browser, 62
Unauthorized status error (401), 55, 62
unhandled exception, in JavaScript, 100
Unicode, 33
  decimal *&#number;* notation for, 77
  escaping method based on, 113
  JavaScript support, 112
  whitespace, 74n
Uniform Messaging Policy, 240
Uniform Resource Locators (URLs), 23–40
  browser processing, 29–31
  common schemes, 36–38
  constructing based on user input, 40
  encoding, 31
  encoding data in fragment identifiers, 144n
  fully qualified absolute, 24
  hiding with encapsulating protocols, 38
  navigation based on tiers of schemes, 188
  resolution of relative, 38–39
  structure, 24–31
    credentials, 26
    fragment ID, 28–29
    hierarchical file path, 27–28
    indicator of hierarchical URLs, 25–26
    query string, 28
    scheme name, 24–25
    server address, 26–27
    server port, 27
UniformRequest API, 240
University of Illinois, 10
Unix services, listener process, 216n
unreserved characters, in HTML, 32
unrestricted URL scheme, 188
URLs (Uniform Resource Locators). *See* Uniform Resource Locators (URLs)
URL-handling APIs, 133
URL-level proposals, 256–257
url(...) pseudo-functions (CSS), 89
user
  browsing habits, *Referer* header and, 51
  collecting information about interaction, 184
  as security flaw, 14–15
  URL construction based on input, 40
User-Agent request header, 43
user content, markup filter for, 86
user-controlled filenames in *Content-Disposition* headers, 67
user data in HTTP cookies, 67
user interfaces
  browser extensions and, 161
  notifications, 259
  timing attacks on, 222–223
username, in credentials portion of URLs, 26
UTF-7 charset, 78
UTF-8 charset, 33, 206
  in HTTP headers, 50
UTF-16 charset, 78, 206
UTF-32 charset, 78

V

valid scheme names, current list, 24
variables, declaration in JavaScript, 99
VBScript, 96
vbscript: scheme, 37, 169–170
vertical tab, in HTML tag, 74
<video> tag (HTML5), 84, 119, 131
view-cache: scheme, 37
View > Encoding menu, 209
view-source: scheme, 37
Visual Basic, 10, 114, 130
VoiceXML, 236

W

W3C (World Wide Web Consortium), 12, 70
w3m, 10
WAP (Wireless Application Protocol suite), 123
WBXML, 123n
WDP file format, 83
Web, the. *See* World Wide Web
web 2.0, 12–13
web applications
  design issues, 263–265
  vulnerabilities specific to, 262–263
WebDAV, 54
WebGL, 131, 131n
Web Hypertext Application Technology Working Group (WHATWG), 13
WebKit parsing engine, 70n, 242
  character set inheritance, 209
  CORS in, 237, 239
  data: URLs in, 168
  history.pushState() API, 256
  Refresh redirection to *javascript:*, 170
  Strict Transport Security support, 248
  Worker API, 258
  XSS-detection logic, 251
web page, prerendering, 258–259
web storage, same-origin policy mechanism for, 148
WebRTC, 257
WebSocket API, 257
WebSQL API, 258
Western European code page (ISO-8859-1), 50
WHATWG (Web Hypertext Application Technology Working Group), 13
whitelists, 226
whitespace, 74, 92
window.alert() API, 218
window.blur() function, 217n, 220
window.confirm() API, 218
window.createPopup() API, 222
window.focus() method, 220
window handles, 175
window.moveTo() method, 220
window.name property, of frames, 175
window.notifications API, 259
window.open() function, 111, 174–175, 217, 217n, 219, 222
window.print() API, 218
window.prompt() API, 218
window.resizeTo() method, 220
windows
  anonymous, 175
  creating new in browser, 217
  and frame interactions, 174–181
  positioning, 219–222
window.showModalDialog() API, 217
Windows Media Player, 119, 130, 132
Windows operating system, 10, 13
window splicing, 220–221
Windows Presentation Foundation, 134, 136
Wireless Application Protocol suite (WAP), 123
Wireless Markup Language (WML), 123
WMF file format, 83
WML Script (WMLS), and JavaScript, 123
WML (Wireless Markup Language), 123
Worker API, 258
World Wide Web
  browser wars, 10–11, 233
  history, 8–13
  threat of hostile takeover, 131
World Wide Web Consortium (W3C), 12, 70
  creation of, 11
  Microsoft and, 239
worms, 12
WWW-Authenticate header, 62, 63
wyciwyg: scheme, 37

X

XAML (Extensible Application Markup Language), 134
Xanadu, 9
XBAP (XML browser applications), 135–136
XBL bindings, 89–90
X-Content-Type-Options header, 208
X-Content-Type-Options: nosniff header, 203
XDomainRequest API, 239–240, 253
X-Frame-Options header, 179–180, 243

XHTML, 12
  and HTML entities, 78
  minimal fault-tolerance of parser, 73
  named entities, 76
  syntax, 70
XML (Extensible Markup Language)
  and bandwidth, 123n
  binary-only serialization, 123n
  case of tags, 72
  <![CDATA[...]]> blocks, 72, 78, 250
XML Binding Language files, 90
XML browser applications (XBAP), 135–136
XML documents
  browser support, 119–124
  generic view, 120–121
XMLHttpRequest API, 12, 54, 210, 236, 237–238
  httponly cookies and, 150
  same-origin policy mechanism for, 146–148
xmlns namespace, 72, 119
<?xml-stylesheet href=... ?> directive, 88
XML User Interface Language (XUL), 122–123
XMLSocket, TCP connections via, 156
<xmp> tag (HTML), 72
XSRF (cross-site request forgery), 84, 190, 262
  exploitation of flaws, 190
  login forms and, 145–146
XSS (cross-site scripting), 71, 262
  bugs, and password managers, 228
  exploitation of flaws, 190
  filtering, 251–252, 253
XUL (XML User Interface Language), 122–123
XXE (External XML Entity) attack, 76

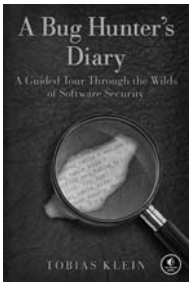
Z

ZIP files, extracting content from, 37
Zone.Identifier metadata, Internet Explorer and, 231
zone model, for Internet Explorer, 229–231

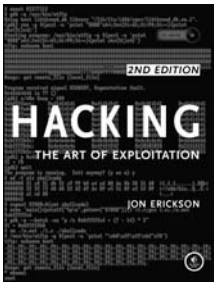
UPDATES

Visit *http://nostarch.com/tangledweb.htm* for updates, errata, and other information.

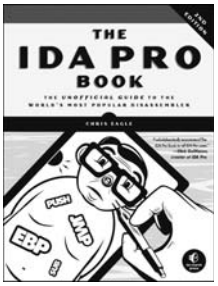
More no-nonsense books from **NO STARCH PRESS**



A BUG HUNTER'S DIARY
A Guided Tour Through the Wilds of Software Security
by TOBIAS KLEIN
NOVEMBER 2011, 208 PP., $39.95
ISBN 978-1-59327-385-9

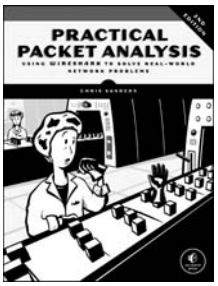


HACKING, 2ND EDITION
The Art of Exploitation
by JON ERICKSON
FEBRUARY 2008, 488 PP. W/CD, $49.95
ISBN 978-1-59327-144-2

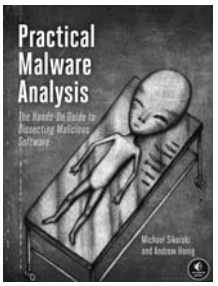


THE IDA PRO BOOK, 2ND EDITION
The Unofficial Guide to the World's Most Popular Disassembler
by CHRIS EAGLE
JULY 2011, 672 PP., $69.95
ISBN 978-1-59327-289-0

METASPLOIT
The Penetration Tester's Guide
by DAVID KENNEDY, JIM O'GORMAN, DEVON KEARNS, *and* MATI AHARONI
JULY 2011, 328 PP., $49.95
ISBN 978-1-59327-288-3



PRACTICAL PACKET ANALYSIS, 2ND EDITION
Using Wireshark to Solve Real-World Network Problems
by CHRIS SANDERS
JULY 2011, 280 PP., $49.95
ISBN 978-1-59327-266-1



PRACTICAL MALWARE ANALYSIS
The Hands-On Guide to Dissecting Malicious Software
by MICHAEL SIKORSKI *and* ANDREW HONIG
JANUARY 2012, 760 PP., $59.95
ISBN 978-1-59327-290-6

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

"Thorough and comprehensive coverage from one of the foremost experts in browser security." – TAVIS ORMANDY, GOOGLE INC.

Modern web applications are built on a tangle of technologies that have been developed over time and then haphazardly pieced together. Every piece of the web application stack, from HTTP requests to browser-side scripts, comes with important yet subtle security consequences. To keep users safe, it is essential for developers to confidently navigate this landscape.

In *The Tangled Web*, Michal Zalewski, one of the world's top browser security experts, offers a compelling narrative that explains exactly how browsers work and why they're fundamentally insecure. Rather than dispense simplistic advice on vulnerabilities, Zalewski examines the entire browser security model, revealing weak points and providing crucial information for shoring up web application security. You'll learn how to:

* Perform common but surprisingly complex tasks such as URL parsing and HTML sanitization
* Use modern security features like Strict Transport Security, Content Security Policy, and Cross-Origin Resource Sharing
* Leverage many variants of the same-origin policy to safely compartmentalize complex web applications and protect user credentials in case of XSS bugs
* Build mashups and embed gadgets without getting stung by the tricky frame navigation policy
* Embed or host user-supplied content without running into the trap of content sniffing

For quick reference, "Security Engineering Cheat Sheets" at the end of each chapter offer ready solutions to the problems you're most likely to encounter. With coverage extending as far as planned HTML5 features, *The Tangled Web* will help you create secure web applications to stand the test of time.

ABOUT THE AUTHOR

Michal Zalewski is an internationally recognized information security expert with a long track record of cutting-edge research. He is credited with discovering hundreds of notable security vulnerabilities and frequently appears on lists of the most influential security experts. He is the author of *Silence on the Wire* (No Starch Press), Google's "Browser Security Handbook," and numerous important research papers.

no starch press

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

"I LIE FLAT." *This book uses RepKover — a durable binding that won't snap shut.*

$49.95 ($52.95 CDN) Shelve In: COMPUTERS/SECURITY

ISBN: 978-1-59327-388-0

9 781593 273880 54995

6 89145 73886 5